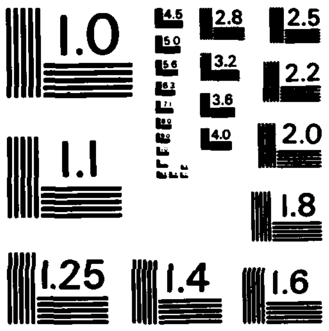
AD-A161 456 ADA (TRADEMARK) REUSABILITY GUIDELINES(U) SOFTECH INC 1/1
WALTHAM MA C AUSNIT ET AL. APR 85 3285-2-208/2
ESD-TR-85-142 F33600-84-D-0280

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ESD-TR-85-142 3285-2-208/2

13

# Ada Reusability Guidelines

CHRISTINE AUSNIT
CHRISTINE BRAUN
STERLING EANES
JOHN GOODENOUGH
RICHARD SIMPSON

SofTech, Inc
460 Totten Pond Road
Waltham, MA 02254

AD-A161 456

April 1985

Prepared for

ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
DEPUTY FOR ACQUISITION LOGISTICS
AND TECHNICAL OPERATIONS
HANSCOM AIR FORCE BASE, MASSACHUSETTS 01731

## REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

[illegible]
ANTHONY L. STEADMAN
Project Officer, Project 2526
Software Engineering Tools and Methods

[illegible]
WILLIAM J. LETENDRE
Program Manager,
Computer Resource
Management Technology

FOR THE COMMANDER

[illegible]
ROBERT J. KENT
Director, Computer Systems Engineering
Deputy for Acquisition Logistics
and Technical Operations

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

AD-A161 456

## REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION: Unclassified | 1b. RESTRICTIVE MARKINGS |
| 2a. SECURITY CLASSIFICATION AUTHORITY | 3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; Distribution unlimited. |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S): 3285-2-208/2 | 5. MONITORING ORGANIZATION REPORT NUMBER(S): ESD-TR-85-142 |
| 6a. NAME OF PERFORMING ORGANIZATION: SofTech, Inc / 6b. OFFICE SYMBOL (If applicable) | 7a. NAME OF MONITORING ORGANIZATION: Hq Electronic Systems Division (AL) |
| 6c. ADDRESS (City, State and ZIP Code): 460 Totten Pond Road, Waltham, MA 02254 | 7b. ADDRESS (City, State and ZIP Code): Hanscom Air Force Base, MA 01731 |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION / 8b. OFFICE SYMBOL (If applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F33600-84-D-0280 |
| 8c. ADDRESS (City, State and ZIP Code) | 10. SOURCE OF FUNDING NOS. |

| PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. | WORK UNIT NO. |
|---|---|---|---|
| | 5720 | | |

11. TITLE (Include Security Classification): Ada* Reusability Guidelines

12. PERSONAL AUTHOR(S): Christine Ausnit, Christine Braun, Sterling Eanes, John Goodenough, Richard Simpson

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (Yr., Mo., Day) | 15. PAGE COUNT |
|---|---|---|---|
| Final | FROM ______ TO ______ | 1985 April | 86 |

16. SUPPLEMENTARY NOTATION: * Ada is a registered trademark of the U.S. Government, Ada Joint Program Office (AJPO)

17. COSATI CODES

| FIELD | GROUP | SUB. GR. |
|---|---|---|
| | | |
| | | |

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Ada
Software reusability

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Reusable software is software that can, with little or no modification, be used in a variety of application systems other than that for which it was originally developed. The Ada language provides many features that support reusability, but reusability can be greatly enhanced if the features are used in certain controlled ways. This report addresses the design, development, documentation, and management issues relating to reusability.

| | |
|---|---|
| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED ☐ [✓ SAME AS RPT.] DTIC USERS ☐ | 21. ABSTRACT SECURITY CLASSIFICATION: Unclassified |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL | 22b. TELEPHONE NUMBER (Include Area Code) / 22c. OFFICE SYMBOL |

DD FORM 1473, 83 APR    EDITION OF 1 JAN 73 IS OBSOLETE.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

## EXECUTIVE SUMMARY

This report presents guidelines for the use of Ada to produce reusable software, that can, with little or no modification, be used in a variety of application systems other than that for which it is originally developed. Reusability offers the potential for great cost savings in DoD systems development. The Ada language provides many features that support software reusability, but reusability can be greatly enhanced if the features are used in certain controlled ways. For example, reusable components must be exceptionally well-tested, and each reusable component must have its own documentation at the level currently required for a CPCI. Reusability also has implications for managers -- the additional cost of developing reusable software must be understood and developers must not be penalized. This report addresses the design, development, documentation, and management issues relating to reusability.

The major motivation for development of this report was support for the acquisition of a highly-reusable Ada implementation of the JINTACCS Automated Message Preparation Systems (JAMPS). JAMPS supports the use of the JINTACCS standard, which will be required in many Tactical $C^3$ systems. Consequently, a reusable implementation has great cost-saving potential. This report can be provided to the JAMPS implementation contractor to be used as a guide during design and development. It also includes guidelines appropriate for government use in managing and monitoring the development effort.

The information in this report, while motivated by the JAMPS requirement, is equally applicable to any development with a requirement for reusable software.

## ACKNOWLEDGEMENT

This document was prepared by SofTech, Inc., under Air Force contract F33600-84-D-0280 (administered by Ms. Pat Knoop, Language Control Facility, Wright-Patterson Air Force Base). Financial support was received from Program Element 64740F, Embedded Computer Systems Engineering and Applications, Electronic Systems Division, United States Air Force Systems Command, Hanscom Air Force Base (ESD/ALSE). The MITRE Corporation provided technical guidance.

## TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

## SECTION 1

## INTRODUCTION

This document provides guidelines for writing reusable software in Ada. It is intended as a practical working guide for software designers and developers, and also as an aid to project managers.

Reusable software is software that can be used in a variety of different overall software application systems. For example, standard mathematics and statistics packages are reusable components in wide use today. Software reusability offers great potential for cost savings. A common function need be programmed, debugged, and documented only once, rather than for every system in which it is used. In addition to the obvious savings of development costs, software reliability increases when already-proven components can be used.

Software reusability is currently a major concern within the Department of Defense. Military systems perform many similar functions, and it is reasonable to attempt to provide common reusable software packages that perform these functions. These are generally much more substantial functions than the simple math library kinds of functions that have traditionally been reused, and the potential benefit of reuse is very great. However, achieving reusability of more complex components such as these is a significantly more complex problem.

The Ada language for the first time makes this problem approachable for the DoD community. Ada offers a number of facilities that support the development of reusable software; indeed, this was one of the major goals in the design of the language. Features such as packages and generics provide the flexibility and customizability required for reusable software, and the mechanisms for expressing such design characteristics effectively.

This guide addresses the writing of reusable software in Ada. t begins by discussing the concept of software reusability and the various aspects of reusability that must be considered (Section 2). References to the literature that may interest the designer of reusable software are then presented (Section 3). The rest of the document consists of the specific guidelines. Guidelines for design, Ada interfaces, documentation, and management are given in Sections 4-7. Throughout the document, specific recommendations are highlighted by enclosing them in boxes. This sets the recommendations apart from general discussion of them and helps the reader focus on the key issues.

It should be noted that the recommendations in this document are *guidelines*, not absolute rules. There are many ways of viewing reusability and even more ways of achieving it. The material presented here does not take the place of the study required to develop a full understanding of a particular design problem, but rather guides the designer in understanding his particular reusability requirements and presents practical advice for meeting these requirements.

# SECTION 2

# REUSABILITY CONCEPTS

Reusability is a term widely used in software engineering literature but its precise meaning is rarely addressed. Yet in order to achieve reusability, it is important to have an understanding of the concepts of reusability. With an understanding of these concepts, the motivation behind the guidelines in this guide becomes much more apparent. This section will address these concepts, in particular:

- o What is reusability? How is it related to portability?
- o What are the problems that reusable software tries to solve, and why are these problems not addressed more often?
- o What are some of the issues relating to the development of reusable software?

## 2.1 *WHAT IS REUSABILITY?*

Whenever an attempt is made to define a term that has been appropriated from general English use for use in an engineering discipline, semantic difficulties can be encountered. In order to solve these difficulties, the following sections approach the definition of reusability from several different viewpoints. The first approach is to present different definitions and analyze them in order to yield an acceptable definition and an understanding of that definition. The second approach is to compare reusability with a term often used interchangeably with it, portability. The final section presents some examples of typical reusable software components, thus allowing a definition by example.

### 2.1.1 Definitions

When developing guidelines for reusability, it is important that all users of the guide have a common understanding of how the term reusability is used by the guideline. The term is defined, implicitly or explicitly, differently in different documents. Each one of these views of the definition is potentially valid for the context in which it is used. However, this guideline will explicitly provide a definition for use.

If one looks in a dictionary for the English language, one is able to derive the following definition.

Reusability

The capability of being used again or used repeatedly.

This definition addresses the core issue of reusability -- the ability to be used again or repeatedly. For the purposes of this guide however, it is not specific enough. There is no indication as to what is being reused or what does it mean to be used or reused.

Rogers(*) in an article on standard architecture for business application programs defines Reuse in the following way:

> "A reusable function is a programming function that is developed once and used in many different programs."

This is closer to what is needed, indicating that what is being considered is a programming function that is used in more than one program. It does not, however, address the functions that are adapted to the new applications, or whether adaptation is even considered necessary.

In attempting to address the issues raised by the earlier definitions, a definition has been developed for use in this guide.

Reusability

> The capability of a software component to be used again or used repeatedly in applications other than the one for which it was originally developed. In order to be effectively reused, the component may have to be adapted to the requirements of the new application.

There are several things to note about this definition.

- The definition is concerned with reuse at the software component level, not at the application level. In general, the things which are to be considered for reuse are individual subprograms or subsystems to perform a well-defined piece of the application. The application as a whole will use that component to help accomplish its overall task.

*Rogers, G. R., "A Simple Architecture for Consistent Application Program Design," IBM Systems Journal, Volume 22, No. 3, 1983.

- The definition does not distinguish between the reuse of top-level components and the reuse of bottom-level components. One form of reuse is for the calling component to be reused, changing or replacing those subprograms which are called. An example of such a situation is where there is a component for doing formatting of the input stream. It must call subprograms to perform the actual I/O. If properly designed, it should work even if it is reused in an application that uses different subprograms to perform the actual I/O. Another form of reuse is for the called component to be reused. The most common example of such a situation is a scientific subroutine library. The components in the library do not change even though they may be called from many different applications.

- The definition makes clear that the component may need to be adapted to the new requirements in order to be effectively reused. When discussing reusability, it is important to note that adaptation is often a necessary part in utilizing reusable software. In general, the adaptation comes on the part of the application which is using the reusable component. The application must adapt itself to be compatible with the requirements imposed by the interface to the reusable component. However, it is sometimes necessary to adapt the reusable component itself to accurately reflect more the requirements of the application. In a totally abstract view, all software can be considered to be adaptable for reuse in all applications; if nothing else, the reserved words in the component can be reused. This is, however, not a practical definition. There is a point at which the cost to develop the software from scratch outweighs the savings from adapting for reuse. Figure 2-1 illustrates the tradeoff.

This definition is sufficiently broad to cover a wide range of situations, yet not so general as to be ambiguous in its meaning. It is this definition of reusability that will be used by this guide.

### 2.1.2 Reusability and Portability

Reusability and portability are terms which are often used interchangeably. Yet they are distinct terms with differing meanings. How different they are depends on precisely how each term is defined. Part of the confusion comes from the fact that particular definitions of each of those terms do overlap in meaning. Although those definitions are appropriate for some situations, for this

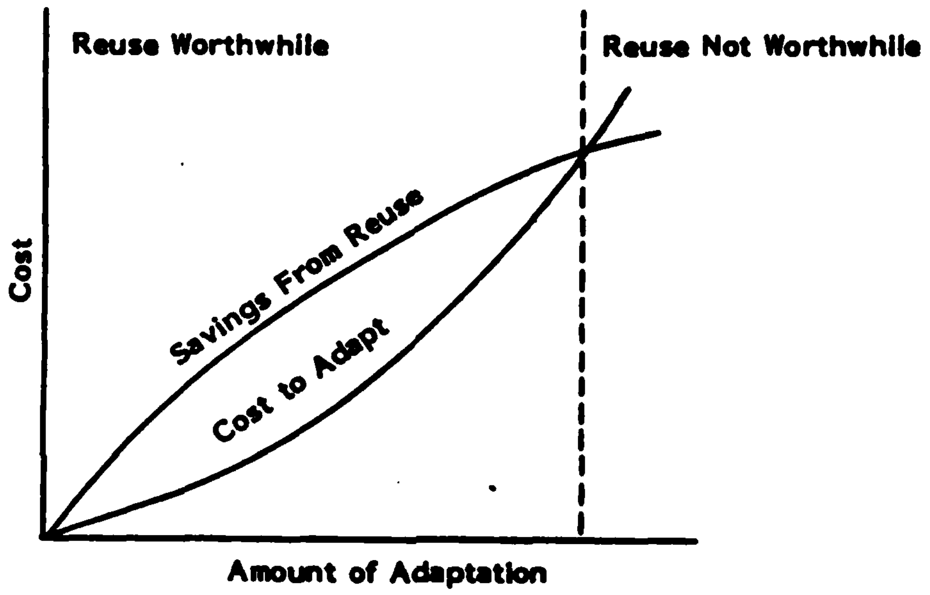


Figure 2-1. Cost to Adapt vs. Cost to Reuse

guide, it is important to make them as distinct as possible. To do that, portability is defined here so that it can be contrasted with reusability.

Portability

> The capability of an application to be used again in a different target environment than the one for which it was originally developed. In order to be ported effectively, the components of the application may need to be adapted to the requirements of the new target environment.

There are several things to notice in the definition of portability that make it distinct from reusability.

- o Portability is normally concerned with transporting an entire application, whereas reusability is concerned with the reuse of a component of that application.
- o When an application is transported, it is used in a new target environment; when a component is reused, it is used in a new application.
- o Portability is concerned with changing the target environment, and any portions of the application which depend on that environment, so that the application reflects the requirements of the new environment.
- o Reusability is concerned with dealing with a different application which uses a component and any aspects of that component that need to change to reflect the requirements of the new application.

Figure 2-2 illustrates these differences.

Another distinction is that, to a large extent, reusability is a design consideration while portability is an implementation consideration. Reusability is achieved primarily by control of the structure of the overall system and of the nature of the interfaces between components. Portability is concerned more with specific use of language features so as to avoid undesired hardware (or other target environment) dependencies. A useful (although somewhat simplified) way to look at this is that reusability deals with software and interface dependencies, while portability deals with hardware and software system dependencies.

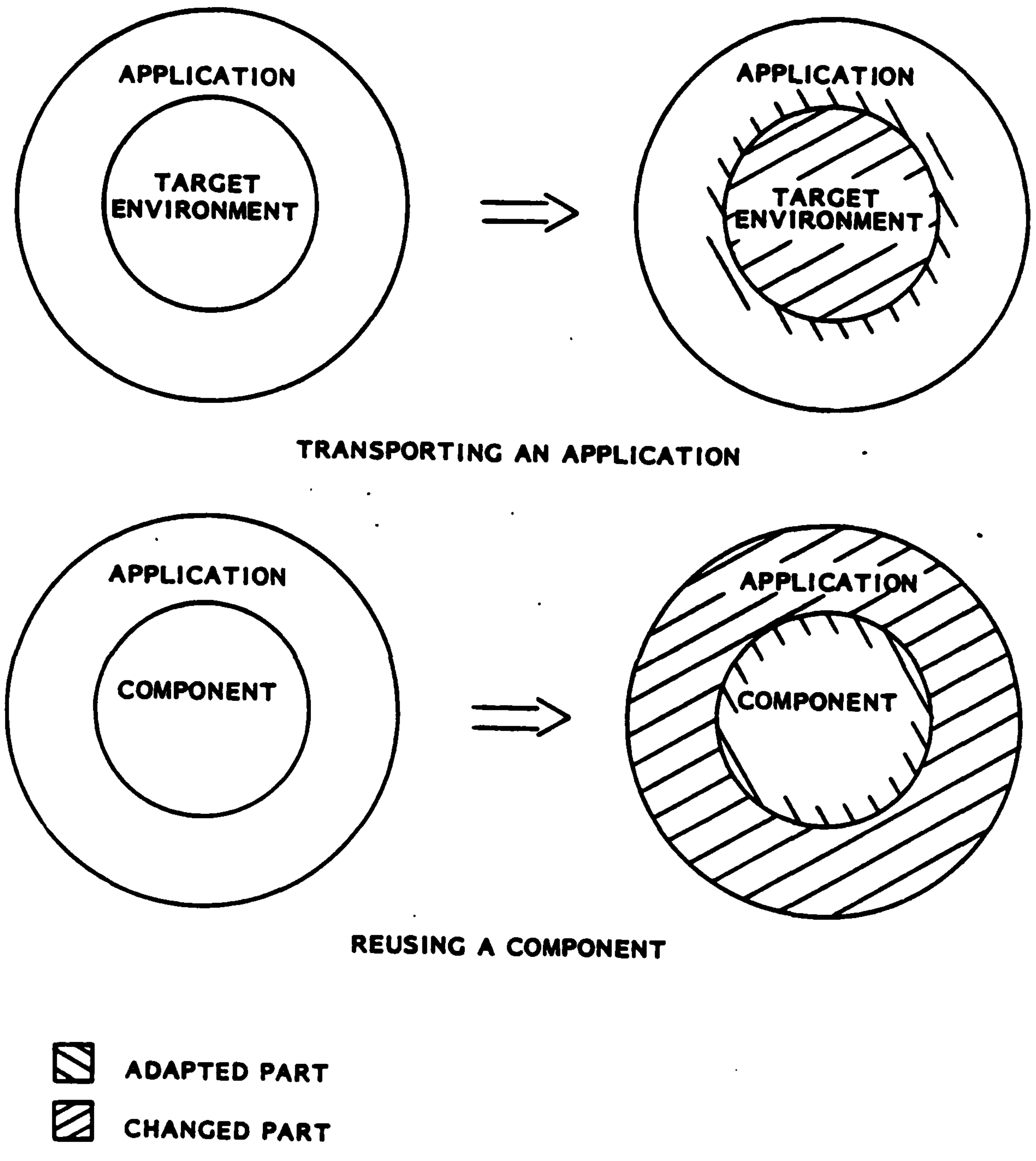


Figure 2-2. Portability vs. Reusability

Portability of an application does not imply that the components of that application are reusable in another application. It is very likely that all components of the application could be tightly coupled, thus preventing any one of them from being used separately in another application. In general, all components of the application will be used in all target environments.

Similarly, reusability does not imply portability. A reusable component could be very target dependent, e.g., an I/O package that is reusable across all projects on one target environment, but unusable in any other target environment.

It is important to note, however, that portability and reusability are not mutually exclusive properties. If it is desired to be able to reuse a component in application systems that run on different target environments, then the component must be designed for portability as well as reusability. Thus, a goal of maximizing reusability will usually include portability as a requirement.

This report deals exclusively with issues of reusability. A companion report on portability,* prepared as part of the same overall effort, discusses portability issues.

### 2.1.3 Examples

Perhaps the best way to understand the meaning of a term is to see some examples of its use. This section looks at several different categories of software, which could be designed for reusability. For each category, its characteristics will be examined to see how it fits within this guide's definition of reusability, and some examples are given.

Support Routines - This category includes routines that are used to support the processing of a function; they are involved in the intrinsic processing of the algorithm.

- o Math and statistics libraries - These classic scientific subroutine libraries provide basic mathematical and statistical functions in a readily usable form.
- o Standard graphics packages - These are packages used by multiple applications to provide a standard interface to a graphics terminal. They provide the basic capabilities of line and shape creation, as well as windowing and other more

*F. Pappas, Ada Portability Guidelines, ESD-TR-85-141, March 1985.

advanced capabilities. Implementations of the Graphical Kernel System (GKS) standard are good examples of this sort of package.

<u>Language primitives</u> - These components provide support for an abstraction which is designed to be usable in a wide range of applications.

- o Data structure support (stacks, lists, queues) - This is the classic example of a reusable component. The component provides abstract support for a complex data structure so users need not be concerned with the details of the implementation.

- o Complex numbers package - This is similar to the data structure support except that it supplies capabilities to make a more complicated data type such as complex numbers appear to the user like a built-in data type, in addition to hiding the details of the implementation.

<u>Program Generators</u> - These tools encourage reusability by allowing the user to specify the requirements for the component of the application, and then having the tool generate the code to actually implement the requirements. A good example of such a tool is a parser generator such as YACC (Yet Another Compiler Compiler available in UNIX). The user specifies the requirements in the form of the BNF grammar. YACC takes the grammar and produces both a table-driven parser and the tables for that parser. Ada generics can be viewed as a somewhat more limited form of program generator where the user specifies the requirements in terms of parameters to the generic template and the compiler creates an instantiation which fulfills those requirements.

<u>Subsystems</u> - These components implement a fairly major portion of the application, and often concerned with the overall algorithm and control of the application.

- o Menu processing subsystem - This sort of system is becoming very common on microcomputer development environments. The vendor provides a standard menu processing subsystem which all applications can use, thus providing a standard user interface to the system.

- o JAMPS message formatting and preparation system - The JAMPS software will be designed to be incorporated within a variety of different applications to provide well-defined interfaces for the preparation and validation of messages, transmission of messages, and reception of messages.

## 2.2 THE IMPORTANCE OF REUSABILITY

The major reason for designing reusable software comes from the expectation that this will increase the productivity of a project that reuses the software component. An article in Strategy for a DoD Software Initiative(*) states that:

> "One way to increase productivity and lower costs is to reuse parts from previous projects in new projects. It should be possible to make parts produced by one DoD contractor available to other contractors, greatly increasing the inventory of reusable parts and the productivity of those who use them."

Other recent studies by NASA have suggested that a factor of four savings in maintenance costs is possible with reusable software.

How is it expected that reusable software components will be able to improve productivity and lower cost? The expectation is that if, for example, in the development of a particular system, that system is composed of 50% reused code, the productivity will have doubled. Effectively, the lines of code produced will have been twice that which would have been possible without the reuse. With increased productivity comes decreased cost. In addition, the reused components should be more reliable than the new code since they will have already been thoroughly tested. However, reuse does not always mean reused components are free. Using code from another application does not come for free. There may be adaptation costs to match the application's requirements to those of the reusable component. The degree of match or mismatch must be examined to determine the efficacy of the reuse. However, in general, incorporating reusable components in an application will increase productivity, improve reliability, and lower cost.

Several issues arise when considering reusing components. The first is how to develop the reusable components in the first place. In general, software is not designed with reuse in mind, because it takes more design and test time to develop reusable components. This potential problem can most effectively be addressed from a

*Anonymous, "Application-oriented Technologies and Reuse," Strategy for a DoD Software Initiative, Vol II: Appendices, 1 October 1982, pp. 186-200.

management side to ensure that there are incentives and milestones to encourage the development of reusable components. Section 7 of this report addresses this issue. The second issue is the impact reusability has on efficiency. A component, having been designed for reusability, will often be more general than is needed for a specific application. Section 4.4 of the report looks at how that impact can be minimized.

Overall, the benefits of reusability are great enough to outweigh these potential problems. In any significant project, it is worthwhile considering reusability, both from the standpoint of investigating the possibility of reuse of existing work and from the standpoint of allowing for the reuse of the software to be developed on the project.

## 2.3 REUSABILITY ISSUES

In order to determine the degree to which effective software reusability can be achieved in Ada, a number of issues must be explored. Several of these issues are summarized below and discussed in detail in the body of this report.

Generality and parameterization. What degree of generality can be built into system components written in Ada? How can components be parameterized to maximize the generality?

Because it will rarely be the case that a major software component can be reused exactly as it is with no changes, it is important for reusability to focus on how generality can be easily achieved. Ada provides generic packages, tasks and subprograms to assist in writing software components that can be explicitly parameterized to allow variability of application. There are also certain design approaches that can be brought to bear to produce software that can be more easily varied. These include the ideas of information hiding and layered design. These design approaches are supported by the Ada package feature. In summary, the use of Ada should make a significant improvement in our ability to produce relatively large reusable software components by its support for component generality and parameterization.

Performance. To what extent can generality be achieved without undue sacrifice of performance?

On the other side of the coin, from variability or generality, is the issue of system performance. In general, it is true that the more detailed knowledge available about the use of a software component, the better it can be made to perform, since advantage can be taken of the specific circumstances of the use. Most experienced programmers can tell horror stories about the poor performance of excessively general systems.

In many cases, though not all, reusability would be worth some sacrifice in performance. The difficult part is determining what is a reasonable tradeoff in a specific case. Ada provides some features that help reduce the cost of added generality. The INLINE pragma, being the most notable, allows small subprograms to have their object code placed directly in the program at the point of their call. This avoids the subprogram call overhead and also allows optimization to occur across the subprogram boundary. A common technique for increasing the generality of a software system is to use small subprograms to "soak up" the differences in different applications. The use of the INLINE pragma allows this technique to be used without requiring a significant degradation in system performance.

Documentation. How can Ada components be documented so a potential user can evaluate them as candidates for reuse?

Clearly, if a software component is to be reused, the potential reusers must know, in detail, what it will do for them. Only if this information is available, can the component be evaluated as a candidate for reuse. Lack of reliable information is a principal contributor to the "not invented here" syndrome that is the often stated reason for not reusing software.

To promote reuse of software, it is necessary to develop a technique for providing information that will be perceived as reliable about the details of the functions provided by a software package. Ada provides some help in this by providing the specification part that specifies the interface to a software component. This does not do the whole job, however, because it does not say enough about the semantics of the component -- what function does it actually do. Some additional conventions, possibly in the form of structured header comments, are required to provide this additional information.

Encouraging Reusable Software. How can customers or management encourage the development of reusable software? How can the incentives be given for development of reusable software?

Like most things in the world, reusable software will not happen by itself. The natural tendency of those responsible for developing a specific software system is setting aside any issues not bearing directly on getting the job at hand done on its schedule and budget. If reusable software is to result from a development effort, it must be stated as an explicit requirement of the job. Only in this way, can a project manager justify spending the time, money, and intellectual effort necessary to make the software reusable.

Managing Reusable Software. How is a requirement for reusability established? How is conformance to requirements verified? Who controls the software once it is developed?

Reusability can only be expected if it is an explicit requirement for a project, established by the customer or by the project manager. However, it is not sufficient to simply state "the software shall be reusable." This immediately raises questions as to scope of reusability (i.e., In what systems? On what different hardware? With what degree of modification?) as well as to how compliance with the requirement is to be verified. The reusability requirement must be more explicitly bounded, and the means for verifying compliance must be established.

Once it has been decided that reusability is a goal, management effort must be expended to ensure the goal is met. Techniques available to help do this include such things as adding reusability questions to the Quality Assurance checklists. It will also be necessary to pay attention to testing the software for the required level of generality.

Since the goal of reusability is to facilitate reuse in multiple applications systems, consideration must be given to the control and distribution of the component once it is developed. A single point of maintenance, configuration control, and distribution must be established. Procedures for other users to obtain the component are necessary. In the case of a government acquisition, GFE issues must be examined. As we move toward a "software components industry," the view of government acquisition and distribution of software components must evolve to support this new technology.

## SECTION 3

## LITERATURE SURVEY

Existing literature on reusabillty is limited. Numerous articles recognize its importance; however, only a small selection discuss it as the primary topic. Our research indicates general agreement on the meaning and goals of reusability. This section will briefly discuss the results of our survey, referencing only those papers making some more interesting points on the topic. A complete bibliography is found at the end of the report.

Reusability is defined as the ability to reuse a module, subsystem or complete system in an application different than the one for which it was originally written. This reuse may or may not involve some modification to the component in order to install it in the new application. The concept of portability is often treated as one of the aspects of reusability. In addition, the following characteristics are commonly found in reusable components:

- communicativeness
- accessibility
- self-descriptiveness
- conciseness
- generality
- modifiability
- simplicity
- machine and application independence
- functional scope

These traits apply to the combination of the code and the documentation that accompanies it. A component should contain a single function, i.e., demonstrate high cohesion and low coupling. The code proper should be well-structured, modular, and appropriately commented. The function coded should be parameterized, where possible, to promote maximum generality, though not at the price of undue complexity. Furthermore, a reusable function should avoid dependence on system hardware or system software, and it should, of course, be meaningful.

The most concentrated collection of related papers is in the special issue of the IEEE Transactions on Software Engineering (September 1984) devoted to software reusability. This special issue contains papers from the "Workshop on Reusability in Programming" which was held in Newport, RI in September 1983. Papers by Horowitz and Munson, Jones, and Standish survey the state-of-the-art with respect to reusable programming. The most relevant papers from the point of view of this report, those by Goguen, Litvintchouk and Matsumoto, and Neighbors, are surveyed below.

In his paper, Goguen describes a number of issues involved in parameterizing programs. Parameterization increases the opportunity to reuse a software module by increasing its flexibility to adapt to the requirements of different applications. He discusses three concepts that go beyond the parameterization provided directly in Ada -- theories, which define the properties required of an actual parameter for it to be meaningfully substituted for the formal parameters of a given parameterized module, views which show explicitly how a module satisfies a given theory, and module expressions which allow modification of modules by adding, deleting or renaming functionality. His approach would be implemented by tools in the Ada Programming Support Environment outside of the standard Ada language processing tools.

Litvintchouk and Matsumoto address the problem of designing an Ada-based software system whose modules will be maximally reusable. Their proposed methodology involves the use of two-level nested generic modules (i.e., generic modules that, when instantiated, yield other generic modules that must themselves be instantiated to get the final modules). They also discuss a method for applying mathematical category theory to the specification of the semantics of reusable module interfaces.

Two studies describe work done on metrics for software reusability. Boehm suggests that metrics be used on each characteristic of the software to compose an overall software quality metric, which can then be used to evaluate the component and its application. Presson discusses a set of metrics to be applied to the different characteristics.

Boehm stresses the importance of the human interface in designing reusable software. Ultimately, a human being must create the inputs and analyze the outputs of many systems. He recommends the use of free-form input to increase the flexibility of use. He also discusses the need for flexibility in the I/O and format definitions to maximize reusability.

Hibbard actually presents a case study on reusable software in his presentation on queues. He discusses all the elements of a reusable package, in particular, the semantics directly contained in the package specification as well as additional semantics, to be specified in the documentation, that have an impact on the correct reuse of the component. He presents the code for a generic blocking and nonblocking queue, discusses the implementation chosen as well as alternate implementation choices. In discussing the alternatives, he explains why those designs were not selected.

Presson's report focuses both on reusability and on interoperability. He analyzes the costs and benefits, concluding that the increased initial costs are soon recovered by the overall long term cost savings. Among the indirect costs, are the inability to foresee all potential reuses and the negative impact on efficiency. There are several categories of reusability, namely module, functional, and system. Within each of these levels, there may be partial or total reuse. The degree of reuse increases with the degree of abstraction embodied in the component. In this report, Presson documents some general guidelines, summarized here. The algorithm should be certified and test data and reports should be made available to the reuser. The implementation language should be standard, not dialect. The code itself should be fault tolerant, well documented, and parameterized. In addition to these recommendations, he suggests that some application areas are more likely to produce reusable components than others. Command and control applications, some support software, and business applications are among the most likely, while missile systems are the least.

Rogers discusses a successful instance of reusable software in IBM's Toronto Laboratory. Business applications were found to share certain common design features which were adapted into a layered architecture. The main programs are identical when viewed as a sequence of initialization, processing and shutdown. The processing, in turn, consists of some combination of well-defined categories, which become a top-level decision logic module. The third level, the detail processing, requires the most individual programming effort. The support and utility functions, which comprise the bottom layer, are accessed by all layers.

Goodenough discusses the reasons that reuse of previously written modules seldom occurs. There are four reasons: descriptive inadequacy, inadvertent loss of generality, overgenerality, and partially met needs. To overcome problems of descriptive inadequacy, an abstract model is helpful. Such a model characterizes common (invariant) capabilities shared by sets of reusable modules. The other problems relate to the design of interfaces. Designing interfaces that place the fewest restrictions on implementation options is difficult in any language, including Ada. The paper formulates an abstract data type and its operations to illustrate a model for a reusable component.

Wegner discusses reusability in a more global context, that of capital-intensive software technology. He discusses the economic justification and explores this technology from four angles: software components, programming in the large, knowledge engineering, and the accomplishments and deficiencies of Ada. He examines the role of abstraction, libraries, and programming development environments in the development of software components. He discusses examples of reusable concepts and products, such as application generators.

Neighbors discusses another approach to reusable software using models. He describes a particular modeling technique using the Draco System. He focuses on the specification and organization of the problem domain and the rules for transformation. Furthermore, he presents examples and experience with the Draco system.

# SECTION 4

# DESIGN GUIDELINES

As noted earlier, reusability is first and foremost a design issue. If a system is not designed with reusability in mind, component interrelationships will be such that reusability cannot be attained no matter how rigorously coding or documentation rules are followed.

In examining the problem of designing reusable software, we find that many of the recommendations simply restate some of the recognized principles of good software design. For example, the principle of abstraction is central to these recommendations. However, this observation does not lessen the validity of these recommendations, but rather provides additional motivation for the recognized design principles.

We have seen that attention to good design principles does not, in itself, lead to reusability, because there are a variety of ways to apply design principles that do not necessarily support particular reusability goals. Designers are confronted with many tradeoff situations where the choice made might depend on an understanding of reusability goals. For example, we examined a well-structured Ada implementation of a message switching application for the possibility of reusing the software that supported creation and interpretation of particular message formats. We found that, for efficiency reasons, this code was too enmeshed in other parts of the application to be reused. The designers made a different choice than they would have made if they had been designing the reusability.

The goal of these design guidelines, is to explain how various design principles support reusability so that the designer can take this into account when making design tradeoffs.

The following subsections discuss several critical issues relating to the design of reusable software. These include the use of models, layered architectures, interface considerations, and efficiency tradeoffs.

## 4.1 MODELS

A model provides a common viewpoint for stating software needs and matching these needs against existing capabilities.

In attempting to increase software reusability, there are two basic problems to be solved:

- o How do you ensure new systems are designed to reuse existing software when appropriate?

- o How do you design software so it is likely to be usable in a variety of systems not yet implemented?

Software will not be reused unless a system is explicitly designed to take advantage of existing reusable software. This means system designers must recognize and take advantage of possibilities for software reuse. On the other hand, if the existing software has not been designed correctly, it will not fit conveniently into any other system, and so will not be reused.

These twin problems can be avoided by having in mind a model for viewing the functionality provided by the software that is to be reused. The model provides a common viewpoint for expressing the needs of a design and the capabilities of existing software, and so increases the probability of finding a match between needs and capabilities. Designing software to fit a model also increases the chance that it will be able to be reused in other systems.

For example, if one wants to create reusable software for manipulating graphic displays, then one first has to define a model of the semantic functions needed to establish and manipulate a certain kind of display interaction. As a model, one might choose as a model the idea that a form will be presented to a display user who will be asked to fill in the fields of the form, following prompts on the display indicating what kind of data is to go in each field.

Even a general descriptive model, such as this fill-in-the-blanks model, helps establish a basis for reusability. A designer of a system can ask himself whether any capabilities needed for his system can be provided within this framework of interactive display processing. If so, he can further consider the detailed functionality provided by software designed within the framework to see if his specific needs can be met. This is an example of how the existence of a general model can influence the design of a system so existing software modules can be reused.

For the person providing the modules, the potential for reusability will be increased if the major needs of possible future users are met by the functionality provided by the modules. Equally important, software use does not make it impossible to meet some requirement. For example, the Ada TEXT_IO package assumes that after opening a text file, the current line, page, and column numbers are all one. This assumption precludes opening a file with the intention of appending data to the end, even though such capabilities are supported by many operating systems. If having to append to files is important in order to have an efficient program, then Ada TEXT_IO cannot be reused in this context. The design of the interface has made inaccessible a desired and supported capability of the underlying operating system because the Ada model of files did not include the possibility of appending to the end of an existing File.

One example of the use of a model to improve reuse of software is the Graphical Kernel System (GKS) [ISO Draft International Standard 7942, June 1983]. GKS is a conceptual model of a graphics system and a set of function the system supports. Developers of graphics software (and sometimes, hardware) are using this model in deciding what functions to provide. Users of graphics equipment can write programs calling the GKS functions with some assurance that they will operate satisfactorily for a wide range of graphics devices.

GKS is a language independent model. The following kinds of functionality are among the ones provided:

| | |
|---|---|
| POLYLINE | draw a sequence of connected straight lines between a list of points |
| FILL AREA | fill the area defined by a sequence of points with a pattern |

In addition, there are attributes controlling the form of output, e.g., the line width, line type (solid, broken, blinking, etc.), and color.

Since GKS is language independent, a binding to a particular language must be provided to ensure people use the same calling conventions for invoking the GKS functions in that language. A system designer can decide how to phrase his programming needs in terms of the functionality provided by GKS, and then given a binding to a particular language, use the language specific conventions for actually obtaining the functionality. To improve portability and reusability of software, any system using graphics devices ought to consider developing its graphics functions in terms of the GKS proposed standard.

Develop models by generalizing from real problems.

In developing a model to serve as the focus for implementing reusable software, take a real problem, and generalize it by finding a class of problems that it belongs to. For example, the requirements of a specific system might say that forms have to be entered in particular ways. Instead of developing software to satisfy just these immediate needs, generalize the requirements somewhat to include other kinds of form completion. Then, design the software (using the guidelines expressed elsewhere in this document) so it meets the more general requirements, but can still be customized (made efficient) for the specific requirements that existed originally. By using this approach, one can design software that meets a specific short-term need, yet help to ensure the software will still be useful in solving similar problems.

As an additional example, consider a message preparation system that assists in the creation of messages chosen from a specific set of possibilities. Each message has a specific set of rules that determines its form and the allowable contents for its various parts. Rather than define the system to handle exactly the specified set of messages, it would be better to define a model in which we can describe the specific format of particular messages. We then build the system to process any message that can be described by this model. Finally, we supply the description of the specific messages to be processed. This allows the same message preparation system to be reused in other application requiring different sets of messages to be prepared.

The key to making software for a particular system reusable in other similar systems is to establish a model at the outset that encompasses both the immediate and long-term needs. Then, design interfaces that are suitable for the general model rather than for the immediate need.

## 4.2 LAYERED ARCHITECTURE

A layered architecture contributes to reusability by separating concerns into discrete layers that can be separately replaced to tailor the software function and performance.

A layered architecture is a software design in which the system is partitioned into an ordered series of discrete layers, each of which implements a specific abstraction. each layer provides a set of services to the layer above it. it, in turn, uses the services provided by the layer below to carry out its own work. In effect, the layers below a given layer define an abstract machine that is used by that layer to carry out its functions. Each layer has specific, defined interface to the layers directly above and below it in the hierarchy. The general structure of a layered architecture is illustrated in Figure 4-1.

A layered architecture contributes to reusability because it allows individual layers to be independently replaced to tailor the function or performance of the system, thus limiting the effect of a change to a relatively small part of the system. Since it is rarely the case that a moderate to large system can be reused exactly, limiting the effects of the required changes can significantly reduce the effort required to modify the system for reuse.

An example of a layered architecture for a simplified form of data management system is shown in Figure 4-2. The bottom layer, the physical device layer, does the physical I/O to the data storage devices. This may include issuing actual I/O device commands, handling device interrupts, and dealing with device errors. This layer supports the abstraction of an idealized I/O device that simply and reliably reads or writes data blocks from one or more of a large set of possible physical device addresses. If the entire data management system was to be reused in another system that had a different type of physical data storage device, only this layer needs to be changed to deal with the new device.

The next layer from the bottom, the file I/O layer, makes the data storage appear to be a set of named files that may be created, destroyed, read from, or written to. It uses the physical I/O services provided by the physical device layer to read and write directories, file extent blocks, and actual data files. A change in the directory structure, or in the way in which variable length files are handled may be made by changing only this layer.

The third layer, the data access layer, provides access to particular pieces of information. It uses the file I/O facilities to store and retrieve the desired data items. It may arrange, for example, for data records to be stored in alphabetical order according to a particular key field. Modifications to the data access method may be made by changing only this layer.

The top layer, the data manipulation layer carries out specific operations on the data in the database. It uses the data access layer to get to the data. Changes in the way data is manipulated can be made by changing only this layer.

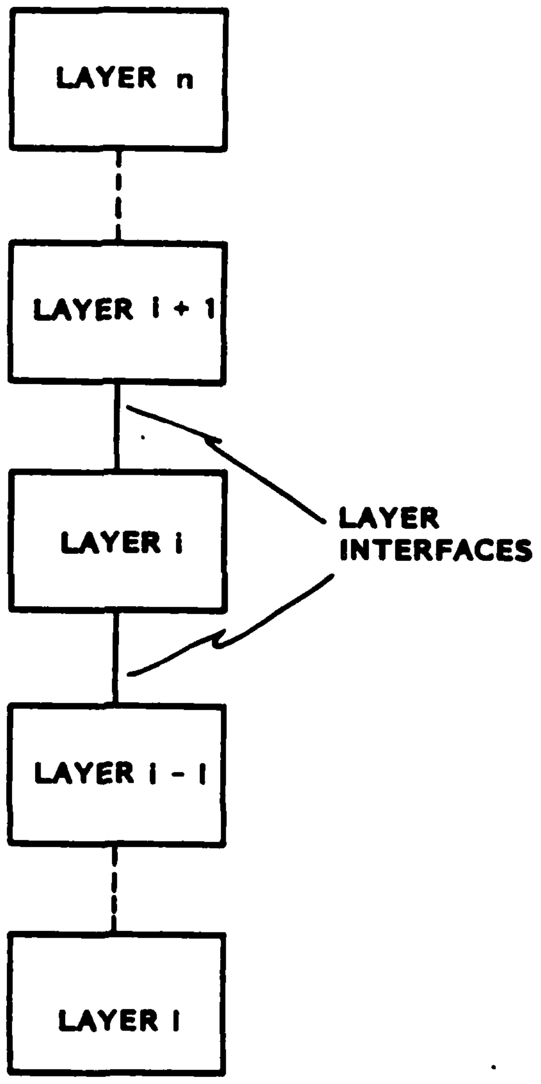


Figure 4-1. Structure of a Layered Architecture

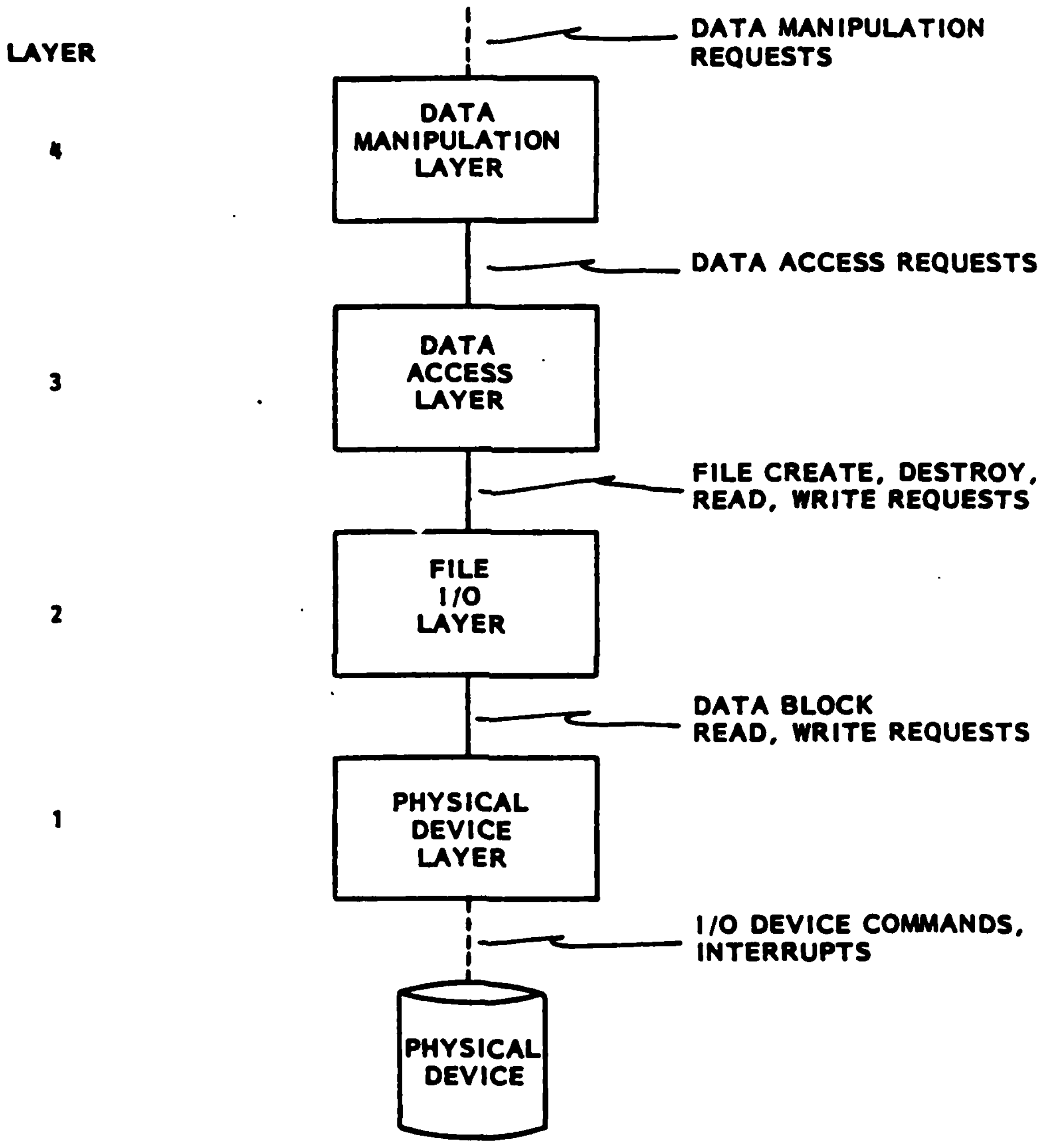

LAYER
DATA MANIPULATION REQUESTS
4
DATA MANIPULATION LAYER
DATA ACCESS REQUESTS
3
DATA ACCESS LAYER
FILE CREATE, DESTROY, READ, WRITE REQUESTS
2
FILE I/O LAYER
DATA BLOCK READ, WRITE REQUESTS
1
PHYSICAL DEVICE LAYER
I/O DEVICE COMMANDS, INTERRUPTS
PHYSICAL DEVICE

Figure 4-2. Layered Database Management System

To reuse this data management system in a different application will typically require some degree of change, most likely to the physical devices supported or to the data manipulations that are done on the data. Either of these changes (and others besides) can be handled by changing only a well-defined part of the system. This reduces the effort and, perhaps more importantly, reduces the opportunity for error involved in modifying the data management system for the new purpose.

It is important to note that, when implementing a particular application using such a general-purpose model, one sometimes ends up with layers consisting of very little processing -- perhaps nothing more than a pass-through to the next layer. This should not be cause for alarm, and any performance penalty should be eliminated by good compiler optimization. Section 4.4 further discusses such optimization issues.

Another example of a layered architecture is the widely used International Standards Organization (ISO) Open System Interconnect (OSI) model of a communications system. This OSI model uses a seven layer architecture to separate the various aspects of the communications task.

A reusable application components become more of a reality, it is reasonable to imagine that generic system architecture models like the ISO/OSI model can be developed for other application areas. For example, perhaps a generic $C^3I$ system architecture could be developed. Such an architecture effectively defines a taxonomy or classification of types of software components. This would provide the standardized functional model essential to component reuse, as well as providing a structure for identifying and managing reusable components. Also essential to reusability, though, is standardization of interfaces between the generic types of components. We can expect the technology to move more slowly in solving this more difficult problem.

An interesting illustration of this sort of evolution occurs in compiler technology. Today there is a fairly widely accepted view of the structure of compilers -- at least most will agree that compilers have a target-independent front end, some sort of "middle" global optimizer phase, and a target-dependent back end, and that there are well-controlled interfaces among these. Simply, the agreement on this structure has allowed for reuse at some level; for example, high-level designs for global optimizations appear in the literature and can be adopted by all implementers because they observe this accepted architecture.

To the extent that the interfaces between the components can be standardized, much greater reuse is possible and has been widely demonstrated. For example, a particular compiler front end is reusable in building a compiler that generates code for the same language but for new target -- the new code generator back end must simply observe the same interface to the front end. However, we have not seen such reuse across development group boundaries (i.e., different vendors) because of the complexity of the interface. The recent effort to define a standard intermediate language, DIANA, for Ada compilers, has tried to take this next step. However, this cross-project reuse has not yet been demonstrated, and some difficulties are anticipated. This is because the DIANA definition probably does not fully define the entire interface between a compiler front end and back end.

One of the most challenging aspects of reusability, is the issue of rigorously defining and standardizing interfaces. The next section examines this issue in more depth.

## 4.3 INTERFACES

Well-defined and documented interfaces are the key to reusable software components.

The interfaces of a software component define the role of that component with respect to the "outside world". A component is completely specified by its interfaces. For the potential reuser of the component, the interface definition is a basis for determining possible component reuse and a guide for using the component. This view is analogous to that of a hardware chip, which can be considered to be specified by its pin connections and the functional descriptions of their roles.

The component reusable designer must analyze and develop a full understanding of the kinds of interfaces that component can have with its users. He must be aware of all interfaces of the component, whether explicit or "understood." There are two reasons for this:

o The designer must consider the interfaces to ensure that they are the right kinds of interfaces to support the degree of reusability that he requires, i.e., that they are necessary and sufficient for his requirements.

o Each interface must be fully documented to permit the user to interface correctly with the component.

The interfaces of a reusable component should be sufficient to provide the necessary flexibility and generality, but should not be so extensive as to limit the potential situations in which the component can be reused.

In general, a component with fewer, simpler interfaces is more reusable than one with a greater number of more complex interfaces. However, the interfaces must be adequate to provide the required generality of function. This can be described as "necessary and sufficient" interfaces, meaning:

- o Interfaces should provide the reuser with the ability to parameterize or customize the component within the intended bounds of reuse. A component with no such interfaces would probably not offer sufficient generality to be of any use. A simple example is a polynomial evaluation function that lets the user specify the order of the polynomial, vs. a function that only handles polynomials of order 4.

- o Interfaces should be limited to those specifically required to support the intended degree of reuse. A large number of interfaces may be indicative of an attempt to be too general or to handle too many special cases. For example, a function that was able to evaluate all possible mathematical expression would almost certainly demand a too-complex interface and too much routine overhead to be of any practical use.

A component must implement a single well-defined function. The intended scope of reuse (i.e., the range of parameterization) must be clearly defined.

Both of these guidelines refer to the "intended degree of reuse" or the "required generality of function." This indicates that a component intended for reuse should implement some single, well-defined functional abstraction. The component designer should define the abstraction to be implemented and should clearly bound the scope within which he expects the component to be reused, and the degree of generality to be supported. He should then define his interfaces with this view in mind. (See the discussion of models in Section 4.1.)

A reusable component should exhibit low coupling and high cohesion.

The concepts of coupling and cohesion are useful in considering the interfaces of a software component intended for reuse. This is simply another way of viewing the points made above. A component with low coupling is one with simpler, fewer interfaces. A component with high cohesion is one that implements a single functional abstraction. Established measures of coupling and cohesion are a way of analyzing the reusability of a component from the standpoint of the necessity and sufficiency of interfaces.

$$\boxed{\text{The interfaces of the component should be explicitly enumerated, classified by type of interface, and considered for necessity and sufficiency.}}$$

As noted above, the designer of the reusable component must carefully enumerate the interfaces of his component, and must consider the degrees of coupling implied by the interfaces. Figure 4-3 is a taxonomy of interface types that may occur between the reusable component and its reuser. The kinds of interfaces are:

a. Reusable component called via subprogram call by reuser.

b. Reusable component calls subprogram of reuser.

c. Reusable component is a task with an entry called by reuser.

d. Reusable component is a task that calls an entry of the reuser.

e. Reusable component shares memory with subprogram of reuser.

f. Reusable component is a task sharing memory with task of reuser.

g. Reusable component communicates with reuser via a shared file with one writing and the other reading.

h. Reusable component communicates with reuser via a shared file with simultaneous access by both.

i. Reusable component communicates with reuser via a message passing or "mailbox" mechanism.

j. Reusable component sends data to and/or receives data from reuser via a communication channel.

k. Reusable component and Reuser have some additional "understanding" about one another's behavior. (This is probably not independent of the other classes, but it may worth calling attention to as a separate dimension.)

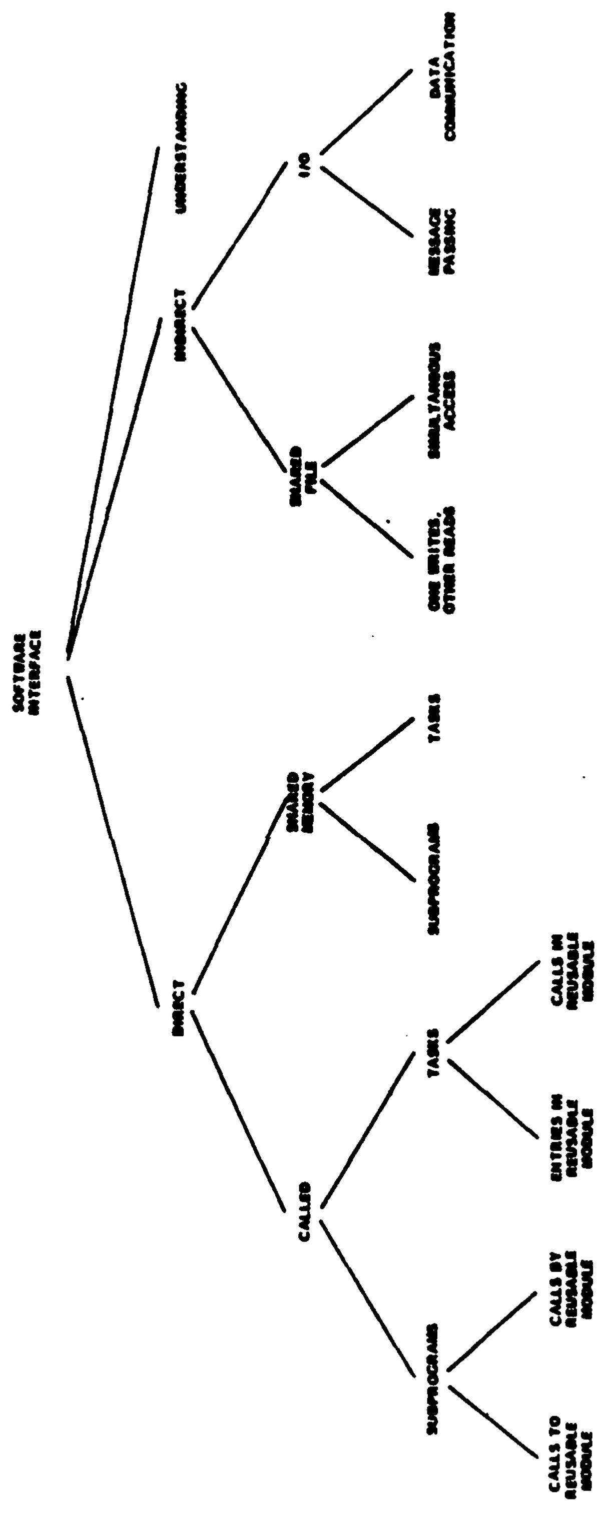


Figure 4-3. Taxonomy of Interfaces

The subprogram and entry call interfaces can be further decomposed to consider modes (i.e., in, out, and in out) and types of parameters.

These kinds of interfaces can be evaluated in terms of the degree of coupling they represent. In general, the more tightly coupled interfaces are shared memory interfaces and shared files with simultaneous access. Interfaces via task interaction are more tightly coupled than subprogram call interfaces. Note that lots of interfaces, or lots of parameters, increase coupling regardless of the kind(s) of interface.

Section 6 gives documentation guidelines for each type of interface.

## 4.4 EFFICIENCY

On first examination, it would seem that designing software for reusability would lead to overly general software with poor performance characteristics. Software designed for reusability will often have capabilities that are unnecessary for a particular user of that software component. These extra capabilities could negatively impact efficiency. However, the proper use of Ada, and the use of a good optimizing Ada compiler can significantly limit the negative impact of these capabilities. This section will address two topics. The first topic addresses the impact that software designed for reusability has on performance. The second topic deals with Ada language and compiler measures that can be taken to minimize this impact on efficiency, without deleting the additional capabilities that are found in reusable components.

### 4.4.1 Areas of Impact

The primary cost in using a software component, designed for reusability, comes from the extra generality that must be provided. A properly designed reusable component will be designed to be usable in as wide a range of applications as is feasible given the scope of the component's design objectives. From the viewpoint of an application using that component, the following sorts of extra generality are often unnecessary.

- o Extra parameters to provide more flexibility
- o Code to check conditions that will never occur
- o Extra subprogram call levels to mask detail

- o Extra subprograms that are only used by some users of the component.

The following paragraphs will examine the necessity of using these features in designing a reusable component. The use of these features will impact efficiency for the user.

Extra parameters to provide more flexibility. Extra parameters for a properly designed reusable component are only unneeded in the sense that the equivalent information could have been encoded into the algorithm of the reusable component with no loss of functionality for the particular application. However, in order to support multiple applications with differing requirements, the reusable component must allow the information to be encoded in a flexible manner. Traditionally, this has been done by adding extra parameters that act as switches to control the algorithm. An example might be a switch to indicate whether the answer should be calculated with extra precision or not. Within a particular application, that decision is generally the same for all uses of the component. This will impact efficiency due to the extra overhead of passing a parameter that never varies.

Code to check conditions that will never occur. For a reusable component to be useful, it must robust. This implies that it will act in a consistent manner even in the presence of anomalous inputs. Quite often, an application can guarantee that these conditions will not occur. Yet in a more limited language than Ada or with a naive Ada compiler, the user would have no means of communicating this to the compiler so that the compiler can take advantage of the information. The presence of this unnecessary code will adversely impact efficiency.

Extra subprogram call levels to mask detail. One issue that often arises in attempting to make use of reusable components is the issue of almost reusable components. These are components that do not quite match the requirements of the particular application. For instance, the types of the parameters might differ or the calling conventions might differ from those used in the rest of the application. One method of addressing the issue of almost reusable components is the use of extra subprogram call levels. These call levels serve to transform the desired use into the available capabilities and vice-versa. This allows the user of a reusable component to use it in the manner that is most appropriate for the application, leaving the task of transforming the use into the form expected by the reusable component to the extra subprogram level (or filter). This will impact efficiency because of the extra call that the filter imposes on all uses of the reusable component.

Extra subprograms that are only used by some users of the component. Quite often, the software design component will lead to the packaging of related capabilities in a single Ada package. For instance, subprograms to calculate the transcendental math functions might be grouped together in a single package both for convenience of packaging and because they might share some common subprograms such as subprograms to evaluate series expansions. However, a user may only care about a limited number of the subprograms. A naive implementation will impose a significant space penalty because it will incorporate all of the subprograms whether or not they are used.

## 4.4.2 Improving Efficiency

Although it is true that designing code for reuse can impact efficiency, the proper use of Ada, in conjunction with a good compiler can limit much of that impact. This section will not attempt to address in detail how to use Ada and the Ada compiler to minimize the impact. That will be addressed more completely in the remaining guidelines sections. It will rather highlight how particular capabilities can ameliorate the impact of the issues raised in the previous section. The capabilities that can help on these issues include the following:

- o generics
- o inline code
- o constant folding and dead code elimination
- o subunits

Figure 4-4 indicates, for each one of these capabilities, the impact that they can have on each of the areas of potential inefficiency. The following paragraphs will look at each capability in more detail.

| Use generic parameters to allow the compiler to optimize a component to the requirements of the particular application. |
|---|

In order to eliminate the impact of extra parameters, it is necessary to be able to communicate to the compiler that this value is constant. This can be done through several mechanisms. If the value is constant for all uses of the component within an application, then making the parameter into a generic IN parameter that is specified with the instantiation eliminates the overhead of passing

| | GENERICS | INLINE | DATIMIZATION | SUBUNITS |
|---|---|---|---|---|
| EXTRA PARAMETERS | H | M | H | |
| CHECKS OF CONDITIONS | M | M | H | |
| EXTRA CALL LEVELS | | H | M | |
| EXTRA SUBPROGRAM | | | | H |

Figure 4-4. Impact of Compiler and Language Capabilities on Generality Imposed by Design for Reuse

what is basically an application-specific constant value as a parameter to the component. In addition, constant folding and dead code elimination can be used more effectively in such situations.

Similarly, checks of conditions can be eliminated through a combination of using generic parameters and dead code elimination. Values which are used to determine the conditions are specified as generic IN parameters. These values are then candidates for constant propagation, allowing the condition to be evaluated at compile time rather than run time. A good compiler will then eliminate the unneeded code.

> Use the pragma inline to allow the compiler to take advantage of conditions of the particular call.

If a value specified as an extra parameter is potentially not constant for all uses, but is constant for each particular use, then it is appropriate to depend on the INLINE code capabilities. These will allow the constant value to be propagated into the inlined code. In general, any time some of the parameters to the call may be constant, the subprogram is a candidate for inlining. Inlining a subprogram, in and of itself, does not have that significant an impact in these situations. However, the use of INLINE in conjunction with a compiler that performs constant folding and dead code elimination can yield major improvements in efficiency.

Inline code is also one of the most effective mechanisms to lessen the impact of extra subprogram call levels. This is particularly effective because, in general, the filter subprogram will be fairly small. Thus, the cost of inlining the call to the filter is small compared with the cost of inlining the entire component.

> When procuring a compiler, examine its optimization characteristics carefully. In particular, determine how effective it is in performing constant folding and dead code elimination.

As can be seen from the previous discussions, the compiler optimization characteristics can have a significant impact on lessening the negative impact of designing code for reusability. In addition, these capabilities can improve code quality in the absence of the other capabilities. For instance, a component designed for reusability might do different things depending on whether the target machine had a 16 bit or 32 bit word size. That can be determined by checking a compile-time constant value in the package SYSTEM. Even a compiler with limited optimization capabilities should be able to use that constant to eliminate unneeded code.

When procuring a compiler, determine what capabilities are available to limit units to be included in the result of a link.

From a strict language semantics point of view, a compiler and linker are required to include all referenced units and their subunits in a link whether every component of those units is needed or not. It is possible to define a linker which includes only those portions of a unit which are strictly needed, thus saving much code space. Therefore, if one has a package of transcendental functions and one chooses to use only the SINE function, an intelligent linker would include only the elaboration code for the package and the code for the SINE function and any subprograms that it references. However, in the extreme this implies that a linker may break apart the results of compiling single units. Although this is possible, it is extremely difficult. It is more likely that a linker will allow the selective inclusion of subunits of a unit. For the case of the package of transcendental functions, this implies that the package be structured so that each subprogram in the package is a separate subunit. The linker will then be able to selectively include those subprograms which are needed by including the subunit for that subprogram.

In conclusion, it can be seen that many of the areas of potential inefficiency can be limited through the proper use of Ada and an effective Ada compiler. In particular:

- o Generics are an effective mechanism for eliminating extra parameters and tailoring code, especially where the information provided by the generic is constant for all uses of the instantiated component. A generic instantiation has many of the benefits of inline code for things which are constant across all uses of the instantiation, without as significant an increase in the required code space.

- o Inline code can have a significant impact on code efficiency. In particular, it will eliminate an extra subprogram call level. Once that is accomplished, the compiler is then able to apply some of the other optimizations that it has available. The disadvantage of using inline code is it can yield a significant increase in the amount of required code space.

- o Constant folding and dead code elimination are capabilities that can have dramatic impact on code efficiency. For reusable software, either generics or inline code must be used to achieve the full impact of constant folding and dead code elimination. These capabilities can also serve to lessen some of the negative space impact of the use of generics and inline code.

- The use of subunits with an intelligent linker can allow a selective linking capability, thus limiting the size of the result of the link to only those components which are absolutely necessary.

When all else has been done, the code can be modified to reflect the requirements of the particular application.

Finally, it should be noted that this section has not addressed potentially the most effective capability for improving the efficiency of reusable components. This section has assumed that the users of reusable components would not want to modify the code. This is not always the case. In many situations, the reusable components will be modified before being used in a new application with differing requirements. This modification can be used to eliminate many of the efficiency problems associated with unmodified components. This may also be an effective strategy when the application must use a compiler that does not effectively support some of the above capabilities, in particular the optimization capabilities. The reuse strategy may be to adapt the code in the way that an optimizing compiler would have adapted it if it could have been used.

## 4.5 LIBRARIES

Traditionally, reusable software components have been made available through software libraries. A software library consists of a collection of subprograms sharing a common purpose. A programmer, wishing to use one of the subprograms, needs to only include a call to the appropriate subprogram in his code. At the time the program is linked, the programmer specifies what software libraries should be searched in order to resolve references to these subprograms. It is generally possible to include multiple libraries, potentially with duplicate entries. The order of search determines which one is used. There are many benefits to the libraries form of reuse.

- It enhances reuse by collecting related subprograms into a centralized location that is readily accessible to the programmer.

- It usually provides well-defined interfaces as part of the documentation of the library.

- It improves productivity by permitting the programmer to focus on the critical aspects of the application.

- It provides a flexible sharing mechanism, allowing the programmer to incorporate any changes to the library simply by relinking.

- The programmer can defer selecting the library to be searched until link time, thus allowing the incorporation of tailored versions of the components. For example, at link time, it is possible to specify whether to include a debug version of a component or a production version.

However, there are some disadvantages with this form of reuse.

- If the library is updated without the programmer being notified, he may get a new version of a component when he is not expecting one.

- For most systems, there is no checking that the call matches the subprogram other than that the names match. Thus, there is no checking of the types and numbers of the parameters for compatibility. In the extreme case, since it is possible to search multiple libraries, a different subprogram from the one desired may be linked in because it is encountered first in the library search.

Ada presents a different view on the sharing of components for reuse. Ada utilizes the program library concept which contains all the units needed in order to link a particular program. How those units become a part of the program library is not addressed as part of the Ada language standard. In the extreme case, it is conceivable that an implementation would require every unit, that is to be a part of a program library, be directly compiled into that program library. In general, implementations will support a more useful view of sharing. As an example, the ALS permits ACQUIREs of units into a program library from another program library. It is expected that components, that are to be shared, will be placed in separate program libraries so they can be acquired by any programmer needing that functionality. Those units would be subject to the same rules with respect to separate compilation that a unit compiled into the library would be subject to. In general, these program libraries containing the reusable components would be collections of related components similar to the traditional software libraries. There are many benefits to this mechanism.

- It enhances reuse by collecting related subprograms into a centralized location that is readily accessible to the programmer.

- It usually provides well-defined interfaces as part of the documentation of the library.

- It improves productivity by permitting the programmer to focus on the critical aspects of the application.

- It ensures that the call to the subprogram and the subprogram specification match; there is a much lower risk of incorporating an unexpected subprogram.

- A properly designed sharing mechanism will allow the shared components to be updated fairly readily. How this can be accomplished will be addressed later in this section.

- An advanced sharing mechanism would allow the selection of versions of a component to be incorporated at link time. How this can be accomplished will be addressed later in this section.

However, there are some disadvantages with this form of reuse.

- Because some of the capabilities described above are not part of the Ada language, it is very implementation dependent how many of these capabilities will actually exist for a particular implementation.

- Since much of the checking is done at compile time, whenever an interface to a unit changes, there is a potentially significant cost for recompilation of affected units.

In balance, however, it can be seen that the Ada mechanism, when supported by reasonable support for a program library in the implementation, can provide an effective mechanism for sharing software for reuse.

Much of this discussion has been predicated on the assumption of some support for a program library above and beyond the minimum capabilities required by the Ada language. In order to support reuse effectively, the implementation must support some form of sharing. Sharing can take several forms, not all of which are mutually exclusive:

- Sharing of the contents of an entire program library.

- Sharing of an individual unit or hierarchy in a program library.

- Automatic update of the sharing program library whenever the shared unit is updated. This may result in recompilations.

- A freezing of the version in the sharing program library regardless of what happens to the shared unit.

- A freezing of the version in the sharing program library, but with some form of notification capability to notify the user of the shared package whenever it is updated.

Each of these sharing forms has its advantages in particular situations. For a particular implementation, it is important to be aware of how sharing is accomplished so that the programmer can plan for the consequences of the implementation.

To select flexible versions of units is another capability that can be useful in some development environments. This is not only an issue for reusable software components, but an issue in general for Ada development. This can be supported by an implementation in one of several ways. They all depend on the implementation being able to provide the equivalent of variations. A variation is a version of a component that is different than the original, but is still valid. For instance, it is possible to have a production variation and a debug variation of a component. The implementation can provide the flexible selection by supporting variations within the program library. This would allow the programmer to select which variation to include at link time. Alternatively, the implementation could support variations at the program library level. The programmer would then select which variation of the program library to use at the time of the link. The sharing mechanisms discussed above could be used to ensure the libraries remain synchronized.

Another capability that could be useful is the capability to limit recompilations to only those strictly needed by the requirements of the change. This is not a major issue for reused software. This is because reused software components will be generally more stable than a unit under active development. In particular, the interface will not change frequently. Most recompilations are engendered by changing an interface.

Ada libraries are not the same as traditional libraries. They are generally more restrictive than traditional libraries in that they impose more checks on components users than traditional libraries. However, these additional checks provide a much higher degree of safety. With appropriate support within the implementation for reusable components, Ada libraries are an effective mechanism that should lead to the cost-effective development of reliable software.

## SECTION 5

## ADA INTERFACE GUIDELINES

This section presents a set of guidelines for building reusable Ada interfaces. The Ada features which support reusability include the different program structuring devices, exceptions, certain data types, and judicious use of parameter modes. Program structure encompasses packages, subprograms, generic units, and subunits. The guidelines on subprograms and exceptions are closely related to those for packages because of the role exceptions play in defining parts of a package specification.

### 5.1 PACKAGES AND SUBPROGRAMS

The abstraction provided by a package should be complete for a given level of a design.

Completeness in a package specification is an important precept both for programming in general and for reusability in particular. A complete abstraction provides the user the full set of operations he needs to manipulate the abstraction. Without the full set, users may find it awkward to reuse the abstraction in different applications. An incomplete abstraction is likely to undergo modifications and enhancements to its specification, necessitating the reuser to modify his application. The final result is extensive recompilation.

A complete abstraction should include the following classes of operations (i.e., subprograms).

- o creation
- o termination
- o conversion
- o state inquiry
- o input/output representation
- o state change

Abstraction relies heavily on the use of private and limited private types in order to give the implementor maximum freedom and to maintain the integrity and consistency of the abstraction. Because of the language restrictions on private types, it is paramount that all of these classes of operations be provided. For non private types, these classes are obligatory to ensure a consistent level of abstraction for the reuser.

The creation function includes both creating and initializing an object. Termination provides a means of ending the life of the object, essentially making it inaccessible in the remainder of its scope. Conversion allows for the change of representation from one abstract type to another. For example, a conversion function would take as its input, the string "catamaran" and produce as its output an object of a variable length string type which evaluates to "catamaran" or vice versa. In this case, the internal representation of the variable length string object is not accessible for user manipulation. State inquiry functions enable the user to determine the state of an abstract object. One such function is basically a read operation; it evaluates the contents of an object, returning, for instance, "catamaran". Other state inquiry functions allow a user to inquire about boundary conditions. Consider an abstract file type. Boundary conditions cover whether the file is empty or whether its maximum capacity has been reached. Other boundary conditions refer to end-of-line and end-of-page states. Non-boundary state inquiry functions return the position in the file (line and column number), the status of the file (open, closed), and the mode of the file (read-only, write-only, read-and-write). Input/output representations are useful for debugging purposes. The objective is not to debug the abstraction but to allow the user to debug his application. For example, printing out an entire stack, he may discover that he has omitted a "PUSH" operation. State change operations allow the user to modify the contents of an abstract object. For example, replace all occurrences of the letter "a" by "y" to yield "cytymyryn."

An abstraction should not be split across several packages. Use of packages should reflect the abstraction represented by the layered software architecture.

It is poor design to split an abstraction across several packages. Completeness does not preclude a layered approach to software components. In designing a reusable navigation subsystem, for example, the sensor interface will differ from one implementation to another. However, this interface can be designed in two layers so that there is an abstract set of functions through which the navigation computations communicate with the actual hardware:

```
package Navigation_IO is
     procedure Get (Sensor  : in Sensor_ID;        -- top layer
                    Reading : out Float;
                    Failed  : out Boolean);
     .
     .
     .
end Navigation_IO;

with System_Sensors;
package body Navigation_IO is

     Altitude_Sensor
          System_Sensors.Altitude_Task_Type;

     procedure Get (Sensor  : in Sensor_ID;
                    Reading : out Float;
                    Failed  : out Boolean) is

          ...
     begin
          ...
          Altitude_Sensor.Get(...);
          Reading := ...;
          if Reading > ... then
               Failed := False;
          else
               Failed := True;
          end if;
          ...
     end Get;
     end Navigation_IO;

package System_Sensors is                      -- bottom layer
     task type Altitude_Task_Type is
          entry Get(...);
          for Get use at 16#0F#;
          ...
     end Altitude_Task_Type;
     ...
end System_Sensors;
```

The Ada distinction between the package specification and the package body is extremely important in light of reusability. The implication is that multiple package bodies can be developed for a single package specification, with the desired body selected when linking the application. (Some Ada environments, such as SofTech's Ada Language System, support this concept through variation sets in the database.) Different versions of a package body can be chosen for different machines. A caveat is in order here. Different package bodies may introduce subtle semantic differences. The predefined package Text_IO appears to be a reusable package. An IBM and a DEC implementation, however, may differ in the file names that they allow and the conditions under which they raise some I/O exceptions.

| Implement bodies of subprograms declared in a package specification as subunits. |
|---|

Subunits enhance the reusability of software in several ways. A subunit is a distinct compilation unit and as such, it forms an object module that can be separately loaded into memory. By stubbing out the subprogram bodies, the programmer may increase the efficiency of the reusable pieces of software because he maximizes the chance that those subprograms that are not called will also not be loaded. From another point of view, declaring them as subunits allows the bodies to be replaced with minimal recompilation. Only the replacement subunit and its dependents need to be recompiled, assuming the specification is not modified. As with packages, different versions of subprogram bodies may be appropriate for different machines.

## 5.2 GENERICS

| Provide user "hooks" for dealing with boundary conditions. |
|---|

In defining reusable subprograms, situations typically arise in which it is not clear what action should be taken. Such situations may seem like error situations, e.g., attempting to store an element in a queue that is full, or attempting to read a file that contains no more elements. In other cases, it may be possible to let the user define the boundary condition. For example, if a function is supposed to justify a "paragraph" so it has even margins, the processing needed to determine the beginning and end of a paragraph might be supplied by a user-provided subprogram, thereby increasing the formatting function's range of applicability.

In general, providing user hooks makes a routine usable in a wider variety of situations, and so makes it more likely to be reused.

There are three techniques for providing user hooks:

o generic formal subprograms (i.e., passing user-defined subprograms to a general purpose routine)

- exception conditions with or without auxiliary routines for getting additional information about the nature of an exception situation;

- output parameters that provide information about whether a routine completed its action appropriately, and if not, why not.

Generics are widely recognized as a means to construct reusable program units. The very nature of a generic unit is reusable; the code found in the template itself is not run. The template must be instantiated, and it is the various instantiations, or reuses of the original template, that are in fact run.

The instantiations of the generic unit allow for customization of the original template. The instantiation allows the user to specify a set of actual parameters, either in the form of subtype indications or subprogram names. Ada allows different classes of generic formal parameters; the particular class used dictates what kind of reusability is appropriate for a generic unit.

At the most specific level, no generic parameter need be specified. In terms of reusability, this is the most restrictive situation, as the reuser cannot tailor the unit at all. The purpose of such a unit is analogous to derived types: it enables the programmer to distinguish between entities that otherwise look identical.

Generic constants allow you to parameterize the configuration of a system.

In using generic parameters, the most restrictive form is to declare a generic parameter to be a constant. This allows a user to specify, for instance, certain system parameters in order to configure a particular system. The bulk of the software developed for the space shuttle's primary avionics system is reusable.* From one flight to another, some modifications and enhancements are coded; however, many of the changes involve changing software parameters such as load factors, atmospheric data, flight initialization data, orbit data, some state vectors, etc. Recompiling the software with these new parameters, of which there are some thousand or so, reconfigures the system for the next shuttle flight.

*Spector, Alfred and David Gifford, "Case Study: The Space Shuttle Primary Computer System", Communications of the ACM, Volume 27, Number 9, September 1984.

A generic (Boolean) constant allows a user to tailor unwanted generality of a reusable component.

The author of a reusable program is likely to include all sorts of checks to ensure that the program does not accidentally crash. For example, there may be checks on expressions for division by zero, for exceeding some bounds like minus pi to plus pi, for checking that a queue is empty or full, for hardware read errors when reading from disk, etc. Depending on the instantiation, these checks may not apply because the user may be able to show that the case being checked for could never arise. An optimizing compiler may not necessarily recognize such checks as dead code. In order to "help" the compiler remove those checks and make the resulting code more efficient such pieces of code can be executed conditionally, based on the value of a flag which is passed in as a generic formal constant.

Move "constant" parameters out of a subprogram call and make them into generic parameters.

A generic formal constant can be used to define an optional part of the functionality of some subprogram. Many subprograms contain optional processing whose execution depends on the value of some input parameter or flag. Over many calls to this subprogram the actual parameter may have the same value, in effect acting like a constant. This subprogram may be made generic by eliminating this parameter from the specification and listing it as a generic formal parameter, specifically, a generic formal constant. Such a modification improves efficiency because it allows the compiler to remove the unneeded code.

Provide generic formal subprogram parameters as user "hooks":

- If the user can help specify a boundary condition;
- If a situation is encountered that would prevent successful completion of an action and the user may be able to fix the situation so processing can continue.

A default subprogram should be provided to handle the "normal" case.

The most general technique for providing a user hook is to specify a reusable routine as a generic unit with formal subprogram parameters. The formal subprogram is called when a special situation is encountered for which the user can provide helpful advice. For example, a routine for justifying the text of a paragraph could he paragraph, based on the current position of a pointer, using user-provided rules for determining these boundaries. A generic formal subprogram parameter can be declared for this purpose, e.g.,

```
     -- default method for finding paragraph boundaries
         Default_Find_Paragraph_Boundaries
              (Current_Position       : in  ...;
               Beginning_of_Paragraph : out ...;
               End_of_Paragraph       : out ...);
     generic
          with procedure
              Find_Paragraph_Boundaries (Current_Position : in ...;
                      Beginning_of_Paragraph : out ...;
                      End_of_Paragraph       : out ...)
              is Default_Find_Paragraph_Boundaries;
     procedure Justify_Paragraph (Current_Position : ...);
```

The body of Justify_Paragraph will call Find_Paragraph_Boundaries at the appropriate point. A routine is also provided that uses the "normal" definition of how paragraph boundaries are determined. This routine will be invoked if no routine is explicitly provided by the user when Justify_Paragraph is invoked.

As another example, the proper response when encountering the end of volume mark while writing a tape might be to mount another volume and continue writing. An alternate response might be to terminate processing. Since one possible response to this situation requires taking some action and then continuing at the point where the situation was encountered (i.e., continue writing tape), it is reasonable to allow for a user-defined subprogram that takes appropriate action. For example:

```
     generic
          with procedure End_of_Volume_Processing is
              Default_End_of_Volume_Processing;
          ...
     package Record_IO is
          procedure Write (...);
          ...
     end Record_IO;
```

Since one user option is to discontinue processing, the designer of the reusable routine must decide how a user is to indicate this. There are basically two ways: End_of_Volume_Processing can raise a specific exception (e.g., Record_IO_Error) if no further attempts to write are desired; or End_of_Volume_Processing can be declared as function whose return value indicates whether processing is to continue or not.

If End_Of_Volume_Processing raises an exception, then the exception should be passed through to the caller of Write. If Write needs to do some cleanup before passing the exception on to its caller (e.g., if it needs to release some space), the programmer can write:

```
begin
     End_of_Volume_Processing;
exception
     when Record_IO_Error =>      -- user wants to stop
                                  -- processing
                                  -- prepare to leave Write
          raise Record_IO_Error; -- pass the exception on
end;
```

A similar approach would be used if End_of_Volume Processing were a function whose return value indicates whether writing is to continue or not.

## 5.3 ERROR HANDLING AND EXCEPTIONAL CONDITIONS

For each assumption a subroutine depends on to operate correctly, define an exception that is to be raised when the assumption is violated.

Instead of defining a routine such that its user is responsible for ensuring that certain assumptions are satisfied, specify exceptions that are raised when these assumptions are violated. For example, instead of saying "Don't call the Stack.Pop function if the stack is empty", say, "Stack.Pop raises Stack.Empty if called when the stack is empty". Raising an exception allows the user to decide what to do about the situation. In essence, the usability of the routine is extended to include empty stacks.

Violating this Guideline

Some assumptions are expensive to check. For example, a lookup routine that uses binary search must assume the table being searched is sorted. If it is not sorted, the search will return incorrect results. Checking that the input table is sorted is expensive, and since the reason for using binary search is because it is fast, making such a check would be contrary to the purpose of using this search method.

In short, it is not always reasonable to check every assumption underlying the correct behavior of a subprogram.

For every exception declared by a package, define a function that indicates whether if the exception would be raised.

Exceptions are raised to indicate the existence of conditions that prevent a routine from producing its normally expected result. A package of routines is more reusable (i.e., usable in a wider variety of situations) if functions are specified that check for the presence of exception situations in advance of their occurrence. For example, a stack package should declare the exception Stack.Empty, which is raised when Stack.Pop is called for an empty stack. In addition to this exception, a function should be declared, Stack.Is_Empty, which returns True if the Stack is empty. Similarly, a file handling package should provide the function if_ End_Of_File as well as the exception End_of_File to indicate when there are no more items to be read.

These functions are useful to control program logic, e.g., for specifying a guard of an accept statement:

```
select
     when not Stack.Is_Empty => -- closed if nothing is
                                -- on stack
     accept GET (...) do
          Item := Stack.Pop;    -- no exception will be raised
     end Get;
  or
     when not Stack.Is_Full =>  -- closed if stack is full
     accept PUT (...) do
          Stack.Push (Item);    -- no exception will be raised
     end Put;
end select;
```

If such functions are not available, it is awkward to provide the equivalent capability by just using exceptions.

## Violating the Guideline

Functions should not be provided when the amount of computation needed to decide whether the exception will be raised is tantamount to performing the function that will raise the exception. For example, when writing a matrix inversion routine, an exception should be raised if the matrix does not have an inverse. The best way to discover whether a matrix has an inverse is to try to invert it. In this case, it would not make sense to specify the function, Inverse_Exists, since invoking this function will be equivalent to attempting to find the inverse.

Raise an exception if it is cheap to retry an operation after the user has fixed the problem that caused the exception to be raised. Otherwise, let the user provide a subprogram that may be able to fix the problem.

It is usually more convenient to let a subprogram raise an exception when a problem is encountered rather than to attempt to fix the problem (possibly by calling a user-defined subprogram) and then continue processing. This is the case if it is relatively cheap to retry the operation that has encountered the problem. For example, when writing a tape, if the end-of-volume is encountered, it is quite reasonable to raise an exception, since not much work is lost by calling the Write routine again after a new tape has been mounted. But, there are some cases where considerable computation would be lost if an exception is raised. In such cases, it is better to attempt to fix the problem by calling a user-provided formal subprogram.

When additional information is available describing the nature of an exception situation, provide a subprogram to return all available information.

It is sometimes the case that a lot of information can be provided to a user when an exception situation is encountered. For example, tape drives can signal a variety of error conditions, ranging from lack of a write ring, to parity error, to end of tape, etc. Instead of defining one exception for each possible error condition, it may be reasonable to lump all the situations together in a single exception, e.g., DEVICE_ERROR, and then provide an additional subprogram that can be called to obtain all information available about the reason for raising an exception. Providing a routine is more general than providing a global variable that can be read after an exception is raised since any computation involved in producing the information need not be done unless the user wants the information.

In general, if additional information can be provided about the nature of an exception situation, provide a subprogram that will give this information to the user. This subprogram needs to be defined as part of the package interface.

## 5.4 PARAMETERS AND TYPES

Provide automatic initialization of private types using records.

In designing reusable software, it is desirable to provide safe packages and safe abstractions. When using private types, one way to make the package safer is by guaranteeing that every object a user declares of that type will be initialized. When the implementation is naturally a record type, this guideline is easy to follow as the package author provides default initial values for each record component. For private types whose natural implementation is a scalar or array type, the designer treats the type as the type of an enclosing record component. A default initialization can then be provided as before for this single component record. Although the enclosing record does add another layer of data structure, it is transparent to the user of the original abstraction. An optimizing compiler will probably be able to optimize out the record layer in manipulations of objects of this type in order to make the resulting code more efficient.

> In designing subprogram interfaces, use in out rather than out mode parameters.

When calling subprograms whose specification includes parameters of mode in out, the user should be careful that any actual parameters are initialized prior to the call. The subprogram writer, however, gains implementation flexibility which outweighs the disadvantage cited above. This flexibility results from the ability within the subprogram to read the value of the parameter.

> To defer the representation of an abstract type, use an incomplete type and an access type.

An abstraction being implemented by a package might have several possible representations. In order to allow alternative representations without having to recompile the package specification, use an incomplete type and an access type:

```
package Stacks is
     type Stack_Type is private;
     ...
private
     type Specific_Stack_Type;
     type Stack_Type is access Specific_Stack_Type;
end Stacks;
```

The decision about how Stacks are really implemented is now deferred to the package body, where Specific_Stack_Type is defined. Alternate implementations of stacks can be provided by providing different package bodies. Since the alternative implementations can be provided without modifying the package specification, the Stack implementation can be changed without having to recompile all the programs that use the Stack package.

## SECTION 6

## DOCUMENTATION GUIDELINES

Good documentation is essential to software reusability. Without it, reusability will not happen.

Documentation is important in any software development, but is particularly critical for reusable software. Documentation really determines whether the software will be reusable or not, because it allows the potential reuser to determine whether the component meets his needs and then explains how to use it. There are, then, two main functions of the documentation of reusable software:

- It must explain what the component does in adequate detail to permit the potential user to determine whether it can be used in his application.

- It must provide a full explanation of how to use the component.

If either of these objectives is not met, it is likely that the component will not be reused. It has been observed* that one of the main reasons reusability fails to occur is descriptive inadequacy, in which a user fails to find a component meeting his needs because the description of the module's function does not match his own description of his needs. A precise, complete method for describing reusable software components will help overcome this problem. The need for a clear description of how to use the component is obvious. If the user cannot figure out how to use the component, he will find another alternative, and reusability will not occur.

Reusable software documentation has two main purposes -- to tell what the component does, and to tell how to use it.

*Goodenough, J.B. and Zara, R.V., The Effect of Software Structure on Software Reliability, Modifiability, and Reusability: A Case Study and Analysis, Final Report, SofTech, Inc., Waltham, MA, Contract DAAA 25-72C 0667, 1974.

The role of the documentation for a reusable software component is analogous to the role of a hardware component's specification. The hardware component's specification will specify the precise function of the component and will fully describe how it is to be connected in to the rest of the system. The precise meaning of each pin connector will be given. If the chip is user-customizable, instructions for this will be provided. The documentation may say something about the internal function of the chip if that may be of significance to the user, but it will not provide detailed design information.

| Documentation for the reusable component must be complete and self-contained. |
|---|

The documentation for a reusable software component must stand alone, providing everything the potential user or users needs to know. Again, the analogy to a hardware chip is relevant. The user of the chip cannot be expected to delve into the chip's design description, or to call up the designers with questions, in order to understand what it does and how to use it. Similarly, the user of a software component cannot be expected to study the component's design or make contact with its designers. (Today, reuse of all but the simplest sort typically requires such study and personal contact when it occurs at all.)

| Reusability documentation is distinct from the normal required documentation for a software component, though it may share information. |
|---|

Reusability documentation has two components, meeting the two purposes described above. These are the functional description of the component and the interface description. Typically these two components would occur in the same document -- a Reusable Component User's Manual. This document should be considered distinct from other documentation that may be required, for example MIL-STD documentation for a component developed on a government contract. However, as the following subsections will illustrate, the documentation required for a reusable component is quite similar to that required in a B-5 software specification, considering the reusable component as a Computer Program Configuration item (CPCI). A B-5 specification describes the function of a CPCI within a system, and presents its interfaces to the rest of the system. This view of a "black box" with well-defined function and interfaces is essentially the view appropriate to documenting a reusable component. However, it is not oriented toward the specific task of reusing the component, and does not contain all the desired information, as may be seen by examining the suggested outline given below. We recommend that, to maximize the likelihood of reusability happening, the separate reusability documentation be developed. It can refer to appropriate sections of other existing documentation that contain the needed data, if desired. At the minimum, the suggested contents given below should be used as a checklist in examining existing documentation to assess its adequacy.

The reusable component must, of course, have design documentation like any other software. If it is being developed on a government contract and intended for reuse on other government efforts, the MIL-STD documentation will be necessary to the other projects that reuse it for inclusion in their own project documentation.

The following two subsections provide specific guidelines on the two components of the reusable component document, the Functional Description and the Interface Description.

## 6.1 FUNCTIONAL DESCRIPTION

**The functional description describes the operational characteristics of the reusable component.**

The purpose of the Functional Description is to describe what the reusable component does. This includes a statement of the general function of the component, and a detailed presentation of operational characteristics. It is, in effect, a product definition for the component.

A major function of the document is to allow the potential reuser to determine whether the component in fact meets his needs. The document also provides other general user information.

Figure 6-1 is a suggested outline for a Functional Description. While other organizations of the material can be selected, this outline should serve as a checklist to determine whether the necessary information is included. The following paragraphs briefly describe the contents of each section.

**Use Figure 6-1 as a checklist on contents of functional descriptions.**

### 6.1.1 Section 1 -- Functional Summary

This section should present a concise description of the basic function performed by the component. The key consideration in writing this section should be the ability of the potential user to determine whether the component meets his needs. Thus, the termi-

FUNCTIONAL DESCRIPTION
SUGGESTED OUTLINE

1. Functional Summary

   1.1 Component Function
   1.2 Scope of Reuse

2. Documentation References

3. Performance Characteristics

   3.1 Sizing
   3.2 Timing
   3.3 Others

4. Known Limitations

5. User Modification/Customization Provisions

6. Partial Reuse Potential

7. Special Design Considerations

8. Error Handling

9. What to do if a Problem Occurs

   9.1 Troubleshooting
   9.2 Who to Contact

10. Examples of Use

Figure 6-1. Suggested Functional Description Outline

nology used should communicate to the largest audience possible. If the function might be known in somewhat different terms by different potential user communities, it is appropriate to describe it in these various terms.

The model concept discussed in Section 4.1 can be used in this description. The functional model implemented by the reusable component should be described in the more general terms of the model. This is an effective method for allowing potential reusers to assess the fit with their needs.

This section also briefly documents the planned or envisioned scope of reuse. That is, what kinds of applications are expected to be able to make use of this component, and what limitations might make certain kinds of reuse unlikely. This is not intended as a complete description of the component's interface characteristics, but simply as a guide to let the potential reuser know whether he should consider the component a possibility for more detailed investigation. He will have to look at the rest of the documentation (including the Interface Description part) to definitely determine whether he wants to use the component.

### 6.1.2 Section 2 -- Documentation References

This section should reference the design documentation (for example, B-5 and C-5 specifications) for the component, as well as any other documentation that may exist. This might include, for example, test plans and procedures documents. This information is included not because the user is expected to use this documentation, but to allow him to satisfy contractual requirements to deliver documentation for his entire system, including this "off-the-shelf" component.

### 6.1.3 Performance Characteristics

This section documents the performance characteristics of the component. This includes sizing, timing, and other appropriate information. This information is critical to the potential user and must be provided.

Sizing information will include memory occupied by the component, disk space requirements, and any other similar information that may be necessary to determine whether the component can be used (for example, stack space requirement).

Timing information can be expressed in a variety of forms depending on the nature of the reusable component. This might include:

- o absolute time required to perform its function(s) given particular kinds of inputs
- o percent of processor time used by this component in a representative application
- o description of the tasking structure of the component in a way that will permit the user to estimate the task switching overhead to be expected in his application
- o frequency with which the component must be called to ensure correct operation (appropriate to certain realtime applications)

Any such timing information must be presented in terms of the machine configuration(s) and Ada language processor(s) used when collecting the data.

### 6.1.4 Known Limitations

This section presents any operational limitations of the component. This would address any restrictions that make the component do something less than the functional summary would lead the user to expect. These limitations might include:

- o kinds of inputs the component cannot handle
- o functional deviations from the commonly held view of the "abstract function" the component implements
- o dependence on particular characteristics of the Ada processor used (e.g., pragmas or runtime capabilities)
- o dependence on particular characteristics of the target hardware (e.g., existence of particular I/O features)
- o known bugs

### 6.1.5 User Modification/Customization Provisions

Many software components are designed not to be reused exactly as is, but rather to be modified or customized by the user in well-defined ways. Others may allow reuse as is and may also permit user customization. This section enumerates the provisions for modification or customization by the user. Each such modification/ customization should be described, including:

- o purpose of the modification/customization
- o how to make the modification/customization to the software
- o how to verify that this was done correctly

This section should include examples (i.e., the specific changes required) of common customizations that users might be expected to make.

### 6.1.6 Partial Reuse Potential

It may be the case that a user may be interested in reusing part of a component only. For example, the user of a communications protocol package may be interested in the Receive portion but not the Transmit portion. This section documents any such envisioned partial reuses and gives directions for accomplishing them.

Certain such partial reuse may be possible by simply including the complete reusable component in the Ada Program Library and depending on the linker to include only called subprograms. This sort of partial reuse, if envisioned, should still be documented. This arrangement may not be desirable if the size of the unused part is very large, as it effectively becomes a part of the new system and must be configuration controlled, etc., even though it is never really used.

### 6.1.7 Special Design Considerations

Ordinarily the user of the reusable component will not need to be aware of the internal design of the component or algorithms used. However, there may be situations where certain design decisions are important. For example, a user might be looking for some mathematical routine that uses a particular algorithm known for its greater efficiency. It is then appropriate to note that the component uses that algorithm.

It is the responsibility of the reusable component designer to determine whether the component has design characteristics of interest to the reuser, and to document them in this section.

### 6.1.8 Error Handling

This should be a description of how the component responds to incorrect inputs or use. It is probably best presented in some sort of tabular format. If there is extensive information to be provided, e.g., lists of error messages, this can be deferred to an appendix.

### 6.1.9 What to do if a Problem Occurs

This section provides user guidance on what to do if the component does not appear to operate as expected. This should include some trouble-shooting procedures@ and perhaps a table of common symptoms and likely causes. There should also be some statement of who to notify in the event of a problem. (This should presumably be the organization responsible for maintenance of the component.)

### 6.1.10 Examples of Reuse

This section should present representative examples of reuse of the component. Its purpose is to illustrate the kinds of applications that might make use of the component and to show broadly how they are structured to do so. It does not give details of the interfaces, which are presented in the Interface Description.

## 6.2 INTERFACE DESCRIPTION

The precise use of each interface must be described.

The Interface Description enumerates the interfaces of the reusable component and describes precisely how each is to be used. It is the definitive statement of how to use the component, i.e., how the user "connects" it to his system.

The Interface Description is organized by interface. It should contain a subsection for each interface, with a description of how to use that interface. For this purpose, each parameter, shared data item, file, etc., is a distinct interface and should be documented as an entity.

The kinds of information to be presented in documenting an interface depend on the type of interface involved. The following paragraphs describe the information required to document each of the types of interfaces listed in Section 4.3.

### 6.2.1 Reusable Component Called Via Subprogram Call by Reuser

- o calling format (in Ada)
- o complete descriptions of parameters (in Ada), plus any additional information needed, i.e., other constraints on parameter values, expected interrelationships of parameter values, etc.
- o complete descriptions of the meaning of each parameter
- o complete description of exceptions that can be raised

### 6.2.2 Reusable Component Calls Subprogram of Reuser

- o description of expected functioning of the subprogram called
- o complete parameter descriptions, as above
- o exceptions that, if raised by the called subprogram, will be handled, and how they will be handled

### 6.2.3 Reusable Component is a Task with an Entry Called by Reuser

- o complete parameter descriptions, as above
- o any required time dependencies as to when the entry is called
- o conditions under which the call will/will not be accepted, and what waits might occur before rendezvous can happen
- o what wait is incurred before the rendezvous completes how the task is terminated

### 6.2.4 Reusable Component is a Task that Calls an Entry of Reuser

- o complete parameter descriptions, as above
- o description of expected functioning of the called task

- timing requirement -- how fast must call be accepted, and how fast must rendezvous be completed
- expected frequency of calls to called task

### 6.2.5 Reusable Component Shares Memory with Subprogram of Reuser

- complete Ada declaration of the shared data item
- for each element of the shared data item, exactly how the element is set and used by the reusable component, and under what circumstances
- as above, describing precisely how each element is expected to be set and used by the reuser

### 6.2.6 Reusable Component is a Task Sharing Memory with Task of Reuser

- all information for task as described in 6.2.3 or 6.2.4 above
- all information for shared data item as described in 6.2.5
- complete description of procedures for synchronizing reads and writes

### 6.2.7 Reusable Component Communicates with Reuser via a Shared File with One Writing and the Other Reading

- complete description of file structure or reference to such a description
- as for description of shared memory interface, description of how the elements of the file are set or used by the reusable component and how they are expected to be set or used by the reuser

### 6.2.8 Reusable Component Communicates with Reuser via a Shared File with Simultaneous Access by Both

- all information described in 6.2.7 for shared file
- description of synchronization mechanism used

### 6.2.9 Reusable Component Communicates with Reuser via a Message Passing or "Mailbox" Mechanism

- complete description of (or reference to) the protocol
- description of messages sent and expected received messages, and circumstances in which each occurs (includes descriptions of actual and expected responses)

### 6.2.10 Reusable Component Sends Data to and/or Receives Data from Reuser via a Communication Channel

- description (typically by reference) of the protocol used
- as above, descriptions of intended transmissions and/or receptions

### 6.2.11 Reusable Component and Reuser Have an Additional "Understanding" About One Another

- prose or other description appropriate to the type of understanding

# SECTION 7

# MANAGEMENT GUIDELINES

**Reusability will only occur through explicit management direction**

Reusability will never happen "by accident." A piece of software not designed with reusability as a goal, will almost certainly have too many limiting assumptions to be of any general use, and will not have the necessary documentation to permit reuse.

Reusability goes beyond the concern of the individual programmer -- it is the concern of those with a global view concerned with future projects and additional applications. It is management's responsibility to identify a requirement for reusability and to see that it is carried out. (Note that "management" can refer to a variety of organizational entities. It may be a department manager recognizing commonality among his department's projects, a corporate manager deciding that a reusable software component will improve his company's competitive position on subsequent efforts, or a government program manager recognizing the potential for reuse from one program to the next.)

**Management must be prepared to pay a price for reusability.**

It is important for management to realize that reusability has a price. Developing reusable software will cost more in initial design and implementation effort, and the resulting software may perform less efficiently. Management must understand both the benefits and the costs of reusable software to make an informed decision about when the benefits outweigh the costs.

Management direction involves both providing incentives for producing reusable software, and determining that the software produced in fact meets reusability guidelines. The first two subsections of this section deal with these topics. The final subsection discusses the selection of tools to support reusability.

## 7.1 INCENTIVES FOR REUSABILITY

**Reusability will not occur automatically; designers and programmers must be explicitly directed and motivated to ensure reusability.**

Designers and programmers will not produce reusable software unless they are explicitly motivated to do so. By "motivated", we mean that reusability must be made an explicit, well-defined, and measurable project requirement, just as functional and performance requirements are explicit, well-defined, and measurable. Because reusability will be a relatively new requirement to most designers, it will also be important to explain what reusability means and why it is required for the particular software component under consideration.

**Reusability requirements must be as well-defined as other program requirements.**

How is a reusability requirement defined? This is best accomplished by describing the specific scope of reusability required for the component, and by determining in advance how compliance with the requirement will be demonstrated. This approach of defining a requirement in terms of tests that must be met is a familiar and effective one, particularly for somewhat "intangible" requirements.

**The requirements specification for a reusable component should identify the specific reusability demonstration the component will be subjected to.**

Explaining the concept of reusability and the reasons it is important is another management requirement. This guidebook will help with understanding the overall concept as well as general motivations for reusability -- the specific importance to the project at hand must be determined by management and then conveyed to designers and implementers. Presented positively, the idea of widespread use of the software should be an incentive to software developers.

**Specific project guidelines, based on the guidelines in this document, must be established.**

Management goes beyond simply formulating a reusability requirement. Specific guidance on procedures to be followed to attain the requirement must be provided, especially for designers who are new to the problem. This guidebook can serve as the basis for such guidance. Because the guidebook addresses a wide variety of reusability requirements, it is necessary at the start of a project to select from the guidebook those guidelines that are important to the specific goals of the project, and to establish this selected set of guidelines as project direction. These selected guidelines should be concisely documented and clearly conveyed to designers and developers.

Quality Assurance should use a reusability checklist to audit for compliance with guidelines.

Guidelines must not only be established; they must be enforced. For this task, management can call on Quality Assurance. A Quality Assurance checklist, based on the selected guidelines described above, should be prepared. This can then be used in regular audits just as other standards checklists are used.

## 7.2 DEMONSTRATING REUSABILITY

Specific reusability demonstrations must be defined.

If reusability is to be regarded as a requirement, it is necessary to have a way of testing that the requirement is met.

Because tests must test something specific, this demands a specific definition of the reusability required, not simply a requirement that the software component be "as reusable as possible." As noted previously, the requirement may actually be specified in terms of the test to be passed. (This apparent circularity is not a problem -- it is really the most precise way of stating the requirement.)

How can reusability be tested? As described earlier, the reusability of a component can be defined in terms of its interfaces. The interfaces and the values permitted for each define the bounds of reusability of the component. An approach to testing the component's reusability is to develop a set of tests that exercise these various interfaces.

Reusability testing is based on a thorough exercise of the component's interfaces.

This approach takes the view of the reusable component as a "black box" with well-defined interfaces and functionality. The testing problem is then like that for any other software product -- tests provide a range of values at each interface and test for appropriate function. The difference in testing a reusable component and a stand-alone piece of software comes from the fact that the reusable component is intended for use within other pieces of software, and typically has a somewhat tighter coupling to this using environment than a stand-alone program would have. Interfaces may include assumptions about shared global data or "understandings" about the surrounding environment. These interfaces, as well as the more straightforward parameter interfaces, must be exercised.

Sometimes it will not be possible for project management to define the contexts in which a component may be reused, but reusability in general may be considered a desirable goal. This is the "as reusable as possible" kind of requirement. Earlier sections of this document describe the general features of a software component that contribute to its degree of reusability. These include:

- o the numbers and kinds of interfaces
- o the degree of generality or narrowness of the component's function
- o other design aspects such as layered architecture

These characteristics can be considered in evaluating the potential reusability of a particular software design, and project guidelines controlling them can contribute to general reusability.

In considering testing for reusability, the issue of reusing the component with other Ada implementations and on other processors than the original ones will arise. This set of reports considers these topics as portability rather than reusability issues, and addresses them in the Portability Guidelines document.

## 7.3 TOOL SELECTION CRITERIA

Good software development tools can support the development of reusable software.

One aspect of the project manager's job is to make sure the software developers have the tools necessary to do their job. Certain special considerations apply to the selection of tools that support the development of reusable software. Certain requirements must be met if the software is to in fact meet reusability guidelines without unacceptable performance penalty. These requirements apply primarily to the selection of compilers and related tools. Other considerations apply to selection of tools that, while not essential, significantly ease the overall development process. The classes of software development tools that must be considered include the Ada compilation tools and other support software tools. The following two subsections discuss these two categories.

## 7.3.1 Compiler Tools

Reusable code can lead to less efficient programs unless the Ada compiler and associated tools provide appropriate optimizations.

Section 4.4 discussed the reasons why the provisions for generality in reusable code can lead to inefficiency and described optimizations that can help overcome these problems. The selection of an appropriate compiler (and associated tools) can be important to a project requiring reusability.

Some of the specific compiler selection criteria to be considered are:

- o How much interprocedural optimization is performed?
- o Is inline code expansion supported?
- o Does the compiler provide optimizations for the kinds of situations that arise due to specialization of functionality? For example, if substitution of parameters leads to an integer defined with range 1..1, will the code generated treat this as a constant? (Note that range checks will still be required on assignment.)
- o Does the compiler permit the code bodies produced for generated for generic bodies to be shared (i.e., generated once and then called like a procedure for other generic instantiations).
- o Will procedures that are not actually called be omitted from the program during linking?
- o Are the following optimizations supported (all become particularly important in optimizing the kinds of code that will result from expansion of parameterized components):
  - o dead code elimination
  - o constant propagation
  - o loop unrolling
  - o loop fusion
- o Is there optimization to handle special combinations of function calls that can arise from expansion?

It is important to note that more than compiler may require consideration when designing a reusable software component. Typically it will be desired to be able to reuse the component on different computers as well as in different applications on the same computer (i.e., portability as well as reusability will be a goal). The designers of a reusable component should attempt to document the optimizations that are particularly important for that component, so that the potential reuser can determine whether his compiler will support the component adequately.

If a potential reuser wishes to reuse a component but has only a compiler that lacks many of the optimizations required to overcome the generality of the reusable component, one option he may consider is to use the reusable component as a starting point, expanding parameters and optimizing at the source code level. This would have the effect of producing a source program that no longer has the generality of the reusable component. This may be a practical approach for components with relatively simple parameterization, but it requires some sophistication. A tool to perform this process (a source code transformation tool) could be developed based on an existing Ada compiler front end.

### 7.3.2 Other Tools

Tools supporting source code management, configuration control, and testing can help in developing reusable software.

One major class of tools supporting reusable software development supports the management of the code components used to configure a system from reusable components. These might include:

- o library manipulation tools
- o tools to allow substitution of different bodies for a given specification (e.g., ALS variation sets)
- o tools to support a catalog of reusable components along with brief descriptions of their scope of reuse

Configuration control tools are also important. The configuration control of a reusable component is like that of a software product -- the component will have a group of users, possibly geographically dispersed, who must be considered. The users must be able to report problems and obtain updates, and the central maintainer must be able to determine which versions are in use.

These requirements demand significant attention to configuration control. The tools provided in a STONEMAN-compliant Ada environment can be very helpful in meeting these requirements.

Finally, tools can be used to support the demonstration or verification of reusability. These might include:

- o Tools to support test execution and tracking.
- o Standards enforcement tools to audit conformance to coding standards that support reusability.
- o Stub generators to create test environments for reuse.

## APPENDIX A

## BIBLIOGRAPHY

Boehm, B. W., J. Brown, H. Kaspar, M. Lipow, G. Macleod, and M. Merrit, Characteristics of Software Quality, North-Holland Publishing Company, Amsterdam, Holland, 1978.

Freburger, K. and V. Basili, The Software Engineering Laboratory: Relationship Equations, Technical Report TR-764, SEL-3, NSG-5123, University of Maryland, May 1979.

Goodenough, J. B. and R. V. Zara, The Effect of Software Structure on Software Reliability, Modifiability, and Reusability: A Case Study and Analysis, Final Report, SofTech, Inc., Waltham, MA, Contract No. DAAA 25-72C 0667, 1974.

Graphical Kernel Systems (GKS), ISO Draft International Standard 7942, June 1983.

Hibbard, P., A. Hisgen, J. Rosenberg, M. Shaw, and M. Sherman, Studies in Ada Style, 2nd Edition, Springer-Verlag, New York, 1983.

Holloway, G. H., W. R. Bush, and G. H. Mealy, Abstract Model of MSG: First Phase of an Experiment in Software Development, Technical Report 25-78. Final Report, Harvard University, Cambridge, MA, ARPA Order No. 3079.3, October, 1978.

Neighbors, J. M., Software Construction Using Components, Technical Report 160, University of California, 1981.

Parnas, D. L., Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems, Naval Research Laboratory, AD/A-043 369, Washington, D.C., June 1977.

Presson, P. E., J. Tsai, T. P. Bowen, J. V. Post, and R. Schmidt, Software Interoperability and Reusability, Final Technical Report RADC-TR-83-174, Contract No. F30602-80-C-0265, Boeing Aerospace Company, July 1983. (Vol I: ADA138477; Vol II: ADA138478)

Rogers, G. R., "A Simple Architecture for Consistent Application Program Design," IBM Systems Journal, Volume 22, No. 3, 1983.

Smith, D. A. Rapid Software Prototyping, Technical Report 187, University of California, 1982.

Spector, A. and D. Gifford, "Case Study: The Space Shuttle Primary Computer System," Communications of the ACM, Volume 27, No. 9, September 1984.

Wegner, P., "Capital-Intensive Software Technology," IEEE Software, Volume 1, No. 3, July 1984.

## DISTRIBUTION LIST

INTERNAL

D10

A. J. Tachmindji

D-36

J. B. Glore

D-45

G. A. Huff

D-46

S. M. Maciorowski

D-60

J. W. Shay
N. E. Bolen

D-63

G. Knapp

D-65

J. H. Galia
F. D. O'Connor
D.D. Neuman
S. W. Tavan

D-66

R. L. Chagnon

D-67

C. J. Carter
D. P Crowsen
H. C. Floyd
E. J. Hammond
G. E. Hastings

D-67 (continued)

R. G. Howe (35)
G. S. Maday
R. L. Micol
R. W. Miller
P. L. Mintz
C. D. Poindexter
S. M. Rauseo
D. A. Spaeth
E. J. Tefft

D-70

E. L. Lafferty
D. A. MacQueen

D-73

J. A. Clapp
S. J. Cohen
W. W. Farr
M. Gerhardt
E. C. Grund
R. L. Hamilton
M. Hazel
R. F. Hilliard
S. D. Litvintchouk
D. G. Miller
R. G. Munck
C. J. Righini
T. F. Saunders
K. A. Younger

D-75

R. T. Jordan
M. M. Zuk

D-77

J. M. Appico
W. E. Byrne
G. R. Lacroix
A. Sateriale

## DISTRIBUTION LIST

INTERNAL (concluded)

D-101

E. K. Kriegel
J. Riatta
L. C. Scannell
K. Zeh

PROJECT

Electronic Systems Division
Hanscom Air Force Base
Bedford, MA 01731

TCRB

B. J. Hopkins
Lt. J. Graves
Lt. M. K. Paniszczyn

ALSE

W. Letendre
Lt. A. Steadman

Langley AFB
Hampton, VA 23665

TAC/TAFIG

Lt. Col. E. Masek

EXTERNAL

Air Force Armament Laboratory
Eglin AFB, FL 32542

C. M. Anderson

EXTERNAL (concluded)

Air Force Space Division
Directorate of Computer Resources
Box 92960 Worldway Postal Center
Los Angeles, CA 90009

Lt. Col. E. Koss, Director

Army Deputy Director
Ada/STARS Joint Program Office
3D-139 (400 AN) Pentagon
Washington, DC 20301

Lt. Col. R. Stanley, USA

Boston University
College of Engineering
110 Cummington Street
Boston, MA 02215

Dr. M. Ruane (15)
Dr. R. Vidale

Language Control Facility
Wright Patterson AFB
Dayton, OH

G. Castor

Naval Ocean Systems Center
Code 423
San Diego, CA 92152

H. Mumm

WIS Program Office
The MITRE Corporation
D Building
Burlington Road
Bedford, MA 01730

Maj. R. Davis
Capt. Saunders

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314 (12)

AFGL/SULL
Research Library
Hanscom AFB, MA 01731